MOSE: Live Migration Based On-the-Fly Software Emulation

Jinpeng Wei
Florida International University
11200 SW 8th Street
Miami, FL 33199
1-305-348-4038
weijp@cs.fiu.edu

Lok K. Yan
Air Force Research Lab / RIGA
525 Brooks Road
Rome, NY 13441
1-315-330-2756
lok.yan@us.af.mil

Muhammad Azizul Hakim
Florida International University
11200 SW 8th Street
Miami, FL 33199
1-305-348-6250
mhaki005@fiu.edu

ABSTRACT
Software emulation has been proven useful in many scenarios, such as software testing, malware analysis, and intrusion response. However, fine-grained software emulation (e.g., at the instruction level) incurs considerable execution overhead (about 8x performance degradation), which hampers its use in production settings. In this paper, we propose MOSE (Live Migration based On-the-fly Software Emulation) that combines the performance advantages of hardware virtualization and the fine-grained analysis capability (comprehensiveness) of whole-system software emulation. Namely, a system can run as normal on a hardware-virtualized platform at near native speed, but when needed, it can be live-migrated to an emulator, not necessarily running on the same physical system, for in-depth analysis and triage; when the analysis is complete, the virtual machine can be migrated back to benefit from full hardware-virtualization again. In this way, the performance degradation is only experienced during analysis and triage. To demonstrate this new capability, we built a proof of concept on-the-fly software emulation system, based on QEMU/KVM and DECAF, the Dynamic Executable Code Analysis Framework. We also perform three case studies: automated kernel panic triage, live-patching a security vulnerability, and on-demand symbolic execution, to illustrate on-demand instruction level analysis.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Error handling and recovery, Symbolic execution.

General Terms
Design, Reliability, Experimentation, Security, Theory.

Keywords
Demand emulation, live migration, software analysis, symbolic execution.

1. INTRODUCTION
Software emulation has demonstrated its strength in many scenarios, such as software testing, profiling, malware analysis, and intrusion response. However, fine-grained software emulation (e.g., at the instruction level) incurs considerable execution overhead (about 8x performance degradation [5]), which hampers its use in production settings. To address this issue, demand emulation has been proposed to dynamically switch a running system between virtualized and emulated executions [6]. This way, expensive software emulation is used only when there is a need for analysis; otherwise, virtualization gives performance guarantees.

However, existing work in demand emulation is analysis-rigid, cannot support closed source guest OSes, and does not support modern virtualization extensions. For example, Ho's architecture [6] used Xen with para-virtualization for efficient execution of guest VMs and QEMU for emulation and taint analysis. It is not an ideal solution though. First, due to the reliance on para-virtualization, it cannot support closed source OSes such as Windows; Second, it is designed for only one type of analysis: taint analysis, but there are many other types of analysis that are also useful (e.g., tracing, instrumentation and symbolic analysis); Third, it forces virtualized execution and emulated execution to run on top of the same hypervisor (i.e., Xen) on the same physical host, which may not be practical for production systems that do not have enough resources (e.g., battery power) to run emulation-based analysis. For these reasons, we call Ho's architecture *in-host* demand emulation.

In this paper, we propose an *out-host* demand emulation architecture, Live **M**igration based **O**n-the-fly **S**oftware **E**mulation (MOSE for short). The idea is to live-migrate a running virtual machine between a platform with full hardware virtualization and another platform that is based on machine emulation. In doing that, a system can run as normal on the hardware-virtualized platform at near native speed, but when needed, it can be live-migrated to an emulator for in-depth behavioral analysis. Once the analysis is complete, the virtual machine can be migrated back to benefit from full hardware-virtualization once more.

Compared with existing work on demand emulation, our architecture has the following advantages: it can support closed source OSes because it uses full hardware virtualization (For example, our prototype of MOSE can migrate a Windows VM from QEMU/KVM to S2E in Section 7.2); it can take advantage of hardware virtualization extensions; it is analysis-flexible because multiple types of program analysis modules (e.g., instruction tracing, taint analysis, and virtual machine introspection) can be added to the emulation-based platform as plugins; it can support the analysis of systems with limited resources (e.g., battery power) in their normal execution environment (e.g., embedded systems) because expensive software emulation is performed on a separate platform.

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3682-6/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818022

The problem of heterogeneous live-migration can be summarized as the problem of ensuring that a software system (the Operating System kernel in particular) that was initialized for and has executed on one hardware configuration (e.g., hardware virtualization) remains correct and consistent when executing in a different hardware configuration (e.g., software emulation) without the initialization step. This requirement can then be separated into three sub-requirements: hardware device equivalence, hardware state equivalence and software state equivalence (See Section 2). We have designed and built MOSE to satisfy these three sub-requirements. We identified and overcame the technical challenges in live migration between a fully virtualized platform (i.e., QEMU/KVM) and an emulated platform (i.e., DECAF). We found mismatches between the representations of guest VM states by QEMU/KVM and DECAF, such as VGA ram size and network interface card.

To demonstrate the efficacy of our approach, we constructed three use cases on top of the prototype implementation. In the first one (Section 5), we developed a kernel panic triage tool that detects kernel panics in the guest virtual machine (VM) from the hypervisor (QEMU/KVM) and automatically live-migrates the guest into DECAF for analysis and recovery. This used a light-weight kernel panic detection tool in KVM, and a more heavyweight triage tool built into DECAF. This separation of duties demonstrates the motivations behind MOSE.

In the second use case (Section 6), we demonstrate the potential for MOSE to be used as a Software Clinic service. In this scenario, a Heartbleed vulnerable server that is executing on a heterogeneous migration enabled KVM instance is live-migrated to a DECAF to be patched. Once patched, it is live-migrated back to KVM for high performance execution. The server continues to serve requests throughout this period with minimal downtime.

In the third use case (Section 7), we demonstrate that MOSE is not analysis platform dependent. We show that we can improve the performance of S2E, a symbolic execution platform, by allowing the virtual machine to execute in KVM and only migrating the state over to S2E when one of S2E's special instructions are detected. This is an extension of S2E's experimental capability where a virtual machine's state can be saved in KVM and restored in S2E.

This paper makes the following contributions:

- We propose live migration based on-the-fly software emulation that (1) combines the performance advantages of hardware virtualization and the fine-grained analysis capability (comprehensiveness) of whole-system software emulators, and (2) decouples the production environment from the analysis environment.
- We validate the feasibility of our approach by building a prototype of live migration based on-the-fly software emulation, based on QEMU/KVM shipped with Ubuntu Linux and the DECAF analysis framework.
- We perform three case studies that demonstrate the application of our approach in kernel panic triage, security vulnerability patching, and on-demand symbolic execution.
- We evaluate the performance implication of our prototype and kernel panic triage tool on normal applications, by comparing the throughput of an Apache web server in different scenarios.

The rest of this paper is organized as follows. Section 2 discusses technical background and related work. Section 3 presents the architecture of live migration based on-the-fly software emulation. Section 4 describes the implementation of a prototype system. Section 5 presents the kernel panic triage case study, Section 6 presents the security vulnerability patching case study, and Section 7 discusses the on-demand symbolic execution case study. Section 8 reports the results of an experimental evaluation of our prototype. Section 9 discusses possible future directions, and Section 10 concludes the paper.

2. BACKGROUND

2.1 Emulation-based Software Analysis

Software emulation is the translation of software written for a different hardware or OS into software that can run on the current platform. For example: QEMU [12] can emulate a SPARC architecture on top of Linux using the x86 instruction set to run Solaris applications. Inside an emulator, each instruction of the guest application is intercepted and translated into the instruction set of the host system and then executed. Modern emulators like QEMU can run an entire guest virtual machine (including the Operating System and applications) as if it is running on the real target computer.

Because of their visibility and control over the execution of guest applications (e.g., emulation of each guest instruction), software emulators have been employed to create powerful software analysis platforms that can monitor, analyze, or change the execution of target OS or applications. For example, QEMU has been extended into multiple analysis platforms, such as S2E [2], PANDA [4], and DECAF [5]. For this project, we choose DECAF and S2E because they have very useful capabilities for our purpose (e.g., live-patching of vulnerable system processes): DECAF supports virtual machine introspection, guest memory access, plugins for instruction tracing and process listing, and customized plugins, and S2E supports symbolic execution and customized plugins. However, the general technique presented in this paper is not limited to any particular analysis platform.

2.2 Live migration

Live migration is a technique that moves a running virtual machine (VM) or application between different physical hosts without disconnecting the client or application. During live migration, the memory, storage, and network connectivity of the VM are transferred from the original host machine to the destination machine. Because of its benefits such as improved performance, manageability, and fault tolerance, live migration has become a feature supported by almost all major virtual machine managers (e.g., VMware VMotion [15], KVM [9], Xen [16], Hyper-V [10], and OpenVZ [11]).

In this paper, we rely on heterogeneous live migration, which is the live-migration of machines between systems of differing architectures and/or hardware configurations. For an abstract software system, heterogeneous live migration can be performed as long as all external dependencies and interfaces remain consistent across the heterogeneous systems. For example, the instruction set architecture of the two systems must be equivalent because otherwise the software instructions will not execute correctly. As another example, the two systems must also have equivalent Application Programming Interfaces (APIs) and Application Binary Interfaces (ABIs). Since the software system in question is an entire virtual machine, the external dependencies include hardware such as the real CPU, memory and even video and network card states.

We have identified three requirements in total: *hardware equivalence, hardware state equivalence, and software state equivalence*. Hardware equivalence requires that the hardware (CPU, memory architecture, network devices, etc.) on both the source and destination systems must have the same input/output behavior. It is important to note that x86 KVM and x86 QEMU are not strictly hardware equivalent because QEMU does not support all of the hardware extensions and model specific features of the many different CPU models. However, this does not pose too much of a problem since most software programs do not use features that are not supported by QEMU.

Hardware state equivalence requires that the internal state of hardware devices (e.g. network interface card buffers) of the source and destination machines must be equivalent (not equal) such that given the same input the same output results.

Software state equivalence requires that the state information for the software being live-migrated must exhibit the same behavior given the same exact inputs in the same exact order and timing despite the fact that they are executing on different machines. We do not require software equivalence (similar to hardware equivalence) since software state is included in the hardware state (e.g., CPU registers, memory, disk, etc.).

In most cases, software state equivalence can be guaranteed by ensuring that the CPU and memory states of the system are equivalent. This is not the case for whole-system live-migration though because some of the internal kernel state has external dependencies on the hardware.

That is, one of the first operations that an operating system kernel performs during boot is hardware discovery and initialization. During this process, all of the connected hardware devices are enumerated and annotated in internal data structures. Additionally, any devices that are to be used are also initialized (e.g., interrupt vectors and memory addresses agreed upon) and this information is also stored within the internal data structures. Therefore, the operating system has a certain expectation of what the underlying hardware is and how it behaves. This expectation is encapsulated into the hardware equivalence and hardware state equivalence requirements. Please note that these requirements are also present in normal live-migration, but modern hardware virtualization technologies such as VT-d and SR-IOV have made these easier to meet. These remain problematic for heterogeneous live-migration.

2.3 Related work

Although emulation-based analysis is very useful, it has one major drawback, i.e., it can incur considerable execution overhead (e.g., 8x performance degradation when emulating at the instruction level [5]), which hampers its use in production settings.

In order to address this shortcoming, Aftersight [3] decouples analysis from normal execution by recording nondeterministic VM inputs in the production environment and then replaying them on a separate analysis platform. Aftersight reduces the slowdown to the production system because a copy of the program under analysis is run on a separate platform. However, Aftersight can be evaded by malware because the emulated analysis environment can be detected by malware. To address his issue, V2E [17] records malware execution using hardware virtualization for transparency, and then replays and analyzes the malware's execution using dynamic binary translation for flexibility and efficiency of in-depth analysis.

Another approach to reduce the overhead of emulation-based analysis is demand emulation, which dynamically switches a running system between virtualized and emulated executions, so that expensive software emulation is used only when there is a need for analysis; otherwise, virtualization gives performance guarantees. One prime example of demand emulation (in-host demand emulation) is developed by Alex Ho et al. [6]. This architecture used Xen and para-virtualization for efficient execution of guest VMs and QEMU for emulation and taint analysis. However, [6] has several limitations, as we discussed in Section 1.

S2E [14] has an experimental feature that supports saving the guest VM states in KVM and later loading them in QEMU, which shares the same goal with us: to improve the guest's performance. However, the S2E saving/loading process is manual while ours is automatic, and we use live migration while S2E does not.

Shadow driver [13] is an operating system mechanism that monitors device drivers and transparently recovers from transient and fail-stop driver failures; as a result, it enables the system to continue to run correctly when device drivers fail. Our kernel panic triage tool (Section 5) differs from Shadow driver in two ways. First, it can recover the guest OS from not only driver failures but also panics caused by bugs in the core kernel. Second, Shadow driver can only handle transient driver failures caused by the environment, but our kernel panic recovery can handle both transient failures and deterministic failures, irrespective of the cause of the failure.

3. THE DESIGN OF LIVE MIGRATION BASED ON-THE-FLY SOFTWARE EMULATION

The architecture for MOSE is illustrated in Figure 1. Our idea is to live-migrate a running virtual machine between a platform with full hardware virtualization (e.g., QEMU/KVM [8]) and another platform that is based on machine emulation (e.g., DECAF [5]). Figure 2 shows a typical flow diagram of live migration based on-the-fly software emulation. A guest VM initially runs on QEMU/KVM, and when it encounters a trigger (e.g., a NULL pointer dereference) during its execution (Step 1), it is live-migrated (Step 2) into DECAF for an analysis and perhaps a repair (Step 3); once these are done, the guest VM is live-migrated back to QEMU/KVM (Step 4) to continue normal execution (Step 5).

4. PROTOTYPE IMPLEMENTATION

In this section, we present the implementation of a prototype of live migration based on-the-fly software emulation (Section 3). Our prototype implementation instantiates the architecture shown

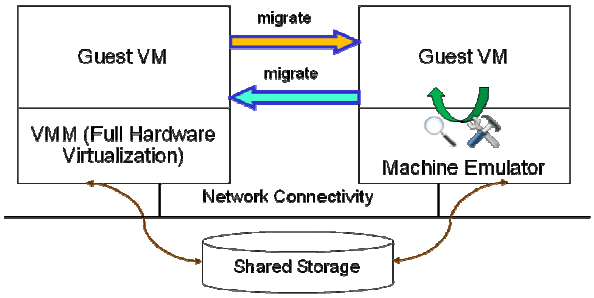

Figure 1. Overall Architecture of Live Migration Based On-the-fly Software Emulation

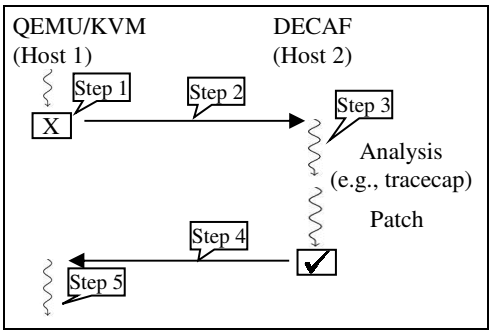

Figure 2. Flow Diagram of a Typical Application of Our Approach using QEMU/KVM and DECAF

in Figure 1. We build the full hardware virtualization platform based on QEMU/KVM [8], and we build the emulation and analysis platform based on DECAF (Dynamic Executable Code Analysis Framework) [5]. Both QEMU/KVM and DECAF are based on QEMU, which reduces the number of possible discrepancies between them and thus eases the live-migration of guest virtual machines between them.

4.1 Overview of challenges

The biggest challenge of a heterogeneous migration presented in Figure 1 is to ensure that the guest VM can continue to execute after it has been initialized for a different environment. For example, both QEMU/KVM and DECAF can handle guest SMIs (System Management Interrupts) if the guest OS is initialized (i.e., booted) from scratch; however, if the guest OS is initialized in QEMU/KVM and then migrated into DECAF, its SMIs cannot be correctly handled because DECAF's handlers for such SMIs are not completely created. This is a fundamental limitation of our approach. In this paper, we discovered and resolved 16 discrepancies to enable live migration between specific versions of QEMU/KVM and DECAF (See Table 1); the relatively short list is partly due to the fact that QEMU/KVM and DECAF are both based on QEMU, which significantly reduces possible discrepancies.

4.2 Internals of normal live migration

To understand the technical challenges, it is helpful to review how live migration works nominally (real implementations migrate part of the VM state prior to suspend stage to minimize downtime). When the source platform and the destination platform run the same software (e.g., QEMU/KVM), a live migration happens in four steps (see Figure 3):

- Suspend: execution of the guest VM is suspended or blocked on the source platform
- Save: the in-memory states of the guest VM are serialized into logical sections to transfer over the network. For example, in QEMU/KVM, the migration state of the guest VM is contained in several sections named "timer", "cpu", etc. (See Figure 3). Details of the "cpu_common" section are also given in (1) the sequence of bytes transferred on the wire, and (2) the type definition of this section. We can see that this section has two fields: one is a 32-bit unsigned integer representing the "halted" field of a memory structure of type "CPUState", and the other a 32-bit unsigned integer representing the "interrupt_request" field of a memory structure of type "CPUState".
- Load: once a section is received by the destination platform, its content is deserialized to restore states of the guest VM in

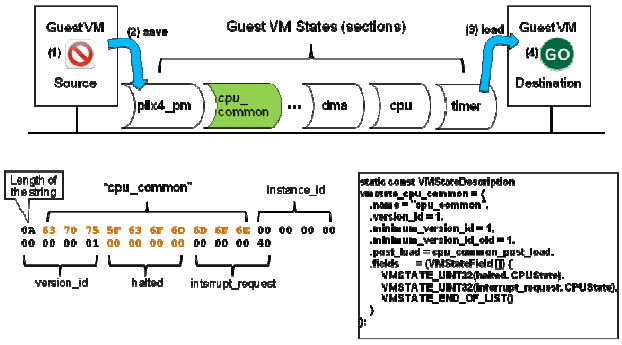

Figure 3. Live migration and guest VM states

memory. For example, the last four bytes of the "cpu_common" section are used to restore the "interrupt_request" field of a guest structure. When all sections are received and loaded, the guest VM's states are completely restored.

- Resume: when the guest VM's states are fully restored at the destination platform, its execution is resumed.

4.3 Possible points of failure when live migrating between different platforms

A migration between QEMU/KVM and DECAF may fail because of two kinds of reasons due to implementation:

There exist discrepancies between their representations of VM states, which cause the "load" step to fail. First, they may support different sets of sections. For example, QEMU/KVM's guest VM state includes a section named "kvmclock", but DECAF does not support such a section, so it aborts the migration with an error message. Second, they may have different type definitions for the same section. Notice in Figure 3 that names of the fields (e.g., "halted") are not included in the transferred bytes, which means that the destination platform must have the same structure definition as the source platform; otherwise, errors may happen. For example, if the destination platform's definition of the "interrupt_request" field is 16-bit unsigned integer, it will restore "interrupt_request" as 0000 instead of 00000040. At a higher level, such discrepancies may be caused by configuration differences, such as supported network interface cards.

There exist discrepancies between their handling of VM states, which cause the "resume" step to fail. Even if the guest VM states are restored successfully, there are differences in how QEMU/KVM and DECAF handle these states because fundamentally QEMU/KVM and DECAF run the guest VM in different ways: QEMU/KVM mainly sets up the environment for the real CPU to run guest code, but DECAF (based on QEMU) directly runs guest code in an emulated CPU. However, an emulated CPU is not exactly the same as a real CPU. For example, DECAF does not emulate the "accessed" bit in the CS descriptor's "Type" field [7], but a real CPU requires that this bit be set, so QEMU/KVM would refuse to run the guest code if this bit in its state is not set. This kind of discrepancy has caused the live migration to fail in our experience.

4.4 Summary of results

Through a careful study and numerous experiments, we identify and resolve 16 discrepancies between QEMU/KVM and DECAF. As a result, we successfully build a prototype of live migration based on-the-fly software emulation. The details of these

Table 1. Discrepancies between QEMU/KVM and DECAF

Discrepancy	Category	QEMU/KVM	DECAF	Resolution
VM state includes the 'kvmclock' section?	HE[2]	Yes	No	Modify DECAF so that it ignores the 'kvmclock' section when accepting a migration
VM state includes the 'kvm-tpr-opt' section?	HE[2]	Yes	No	Modify DECAF so that it ignores the 'kvm-tpr-opt' section when accepting a migration
Handle SMI?	SSE[1]	No, but maintains a spurious SMI flag	Yes	Modify DECAF so that it clears the SMI flag when accepting a migration. Modify QEMU/KVM so that it sets the SMI flag when accepting a migration.
Support the 'conforming' and 'readable' bits?	HSE[3]	Yes	No	Modify DECAF so that it clears the 'conforming' and 'readable' bits in the Task Register's Type field when accepting a migration. Modify QEMU/KVM so that it sets the 'readable' bit when accepting a migration.
Support the 'accessed' bit?	HSE[3]	Yes	No	Modify QEMU/KVM so that it sets the 'accessed' bit in the CS descriptor's Type field when accepting a migration.
PIIX4 PM state includes 'gpe_cpu'?	HSE[3]	Yes	No	Modify DECAF to include 'gpe_cpu' in the PIIX4 PM State
Size of the PIIX4 GPE arrays	HSE[3]	1	4	Change the size in DECAF to 1. GPE: General Purpose Status and Enable arrays
PIT and device assignment support	HE[2]	Yes	No	Configure QEMU/KVM source code to disable PIT and device assignment support
PIT state includes the 'flags' field?	HSE[3]	Yes	No	Modify DECAF so that it ignores the 'flags' field in PIT state when accepting a migration
VGA ram size	HE[2]	16384 * 1024	8192 * 1024	Change the VGA ram size of DECAF to 16384 * 1024
VM state includes the 'DECAF' section?	HE[2]	No	Yes	Modify DECAF so that it does not generate the 'DECAF' section when it starts a migration
VM state includes the 'funmap' section?	HE[2]	No	Yes	Modify DECAF so that it does not generate the 'funmap' section when it starts a migration
VM state includes the 'hookapi' section?	HE[2]	No	Yes	Modify DECAF so that it does not generate the 'hookapi' section when it starts a migration
Needs APIC timer?	SSE[1]	No	Yes, to keep TCG engine alive	Modify QEMU/KVM so that it removes the APIC timer when accepting a migration
Support 'rtl8139' device?	HE[2]	Yes	No	Modify QEMU/KVM so that it uses 'e1000' device, which DECAF supports
Default guest clock source is 'kvm-clock'?	HE[2]	Yes	No	Configure the guest clock source to 'hpet'
Supported clock sources for the guest	HE[2]	kvm-clock, tsc, hpet, acpi_pm	hpet, acpi_pm	

[1]SSE: software state equivalence; [2]HE: hardware equivalence; [3]HSE: hardware state equivalence

discrepancies, which equivalence requirement was violated and our resolution methods are shown in Table 1. The version of QEMU/KVM is qemu-kvm 1.0 shipped with Ubuntu 12.04 LTS, and the version of DECAF is 1.8.

It is important to note that the list of discrepancies in Table 1 may not be exhaustive; there may be other discrepancies but we did not have to address them because QEMU/KVM and DECAF are based on the same QEMU. However, this list is good enough for our tests.

5. CASE STUDY ONE: KERNEL PANIC TRIAGE

In this section, we present the development and evaluation of our kernel panic triage built on live migration based on-the-fly software emulation.

5.1 Motivation

Operating system kernels are designed to be extremely robust against potential errors. Despite the kernel developer's efforts, this robustness can still be compromised by third party software such as device drivers and kernel modules which might not be under

the purview of the kernel developers. Erroneous or buggy drivers can at best lead to kernel panics and at worst lead to privilege escalation vulnerabilities. In a security context, such kernel panics or privilege escalation are advantageous for kernel level malware: for example, kernel panics can be used in a Denial of Service attack. Furthermore, these kinds of errors are difficult to analyze and debug because kernel debuggers can be performance intensive and the rarity of kernel panics makes it even more difficult.

Live migration based demand emulation is well suited for kernel panic debugging because expensive triage (diagnosis or repair) is performed only when there is a kernel panic and in a dedicated analysis environment. Since the triage is not done in the production environment, the production environment can be optimized for the normal (not buggy) case. Therefore, in this section we build a use case of kernel panic triage using live migration based on-the-fly software emulation.

Note that we do not conceal kernel panics from the OS and the applications. Instead, we prevent the system from hanging; we keep the system alive so that important data can be saved (in other words, we let the system survive an otherwise fatal crash). See [13] for a discussion of different kinds of driver errors / faults.

5.2 Design

Figure 4 shows the normal flow of a kernel panic like a NULL pointer dereference. When the kernel execution encounters such an error, the hardware generates an exception, and the exception handler of the kernel is invoked. If the exception handler can fix the problem, it logs debug information by calling *printk* and such information is later flushed to the disk, and then the kernel continues normal execution. On the other hand, if the exception cannot be fixed, the kernel logs debug information by calling *printk* and then stops execution (e.g., panic). In this case, the kernel does not have a chance to flush the debug message to the disk and the only hope is that a human user can see the debug message on the monitor. However, this (i.e. the debug message is visible on the virtual machine's monitor window) may not work when the kernel runs inside a virtual machine.

Motivated by the above basic principle, our kernel panic triage consists of the following steps. We introduce a small and efficient kernel fault detector in QEMU/KVM. When a fault is detected, QEMU/KVM live-migrates the virtual machine into DECAF for further analysis and attempt a repair. For example, when a NULL pointer dereference is detected in QEMU/KVM, we live migrate the VM to DECAF for two kinds of treatments. First, we diagnose the problem by analyzing the kernel's instruction trace and introducing a new plugin that captures the kernel's debug messages (Section 5.3.2). Second, we recover the guest kernel from the panic, by clearing the null-pointer exception state from the VM and cleanly removing the offending kernel module by redirecting guest execution (Section 5.3.3). If the recovery is successful, we live-migrate the virtual machine back to QEMU/KVM for continued operation. The overall design of our kernel panic triage is shown in Figure 6.

5.3 Implementation and Evaluation

In the evaluation of our kernel panic triage, we create a buggy loadable kernel module that under certain conditions writes to virtual address 0, thus triggering a null pointer dereference. We load this buggy kernel module in the guest VM on top of QEMU/KVM to trigger a kernel panic in the guest.

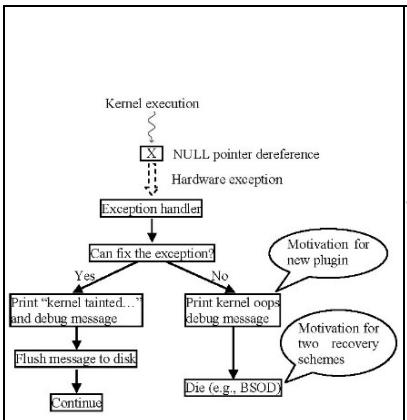

Figure 4. Normal Flow for a NULL Pointer Dereference Exception

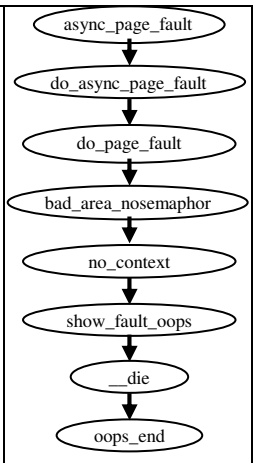

Figure 5. Control Flow of Kernel Oops Handling

5.3.1 QEMU/KVM Fault Detector

We modify the exception handler of the KVM kernel module, so that it passes control back to QEMU with KVM_EXIT_INTERNAL_ERROR when a null pointer dereference happens in the guest VM.

More specifically, a null pointer dereference in the guest kernel will trigger a page fault that is caught by the KVM kernel module. Then the KVM exception handler (i.e., *handle_exception*) checks the faulting memory address in CR2. If the address is 0, the exception handler passes control back to QEMU, setting the exit reason as KVM_EXIT_INTERNAL_ERROR and the sub-error code as 4, and including the value of CR2 in the internal data area of the VCPU's status structure (a data structure used by QEMU and KVM kernel module to exchange information).

We also modified the main loop of QEMU/KVM's execution engine (*kvm_cpu_exec*), such that a live-migration is initiated when KVM_EXIT_INTERNAL_ERROR is seen.

5.3.2 Diagnosis in DECAF

After DECAF receives the guest VM migrated from QEMU/KVM, the first thing an analyst can do is to diagnose the kernel panic. We utilize an existing DECAF plugin to collect the instructions executed by the guest kernel after it is resumed, then we do an offline analysis of the instruction trace to understand the control flow in the guest kernel, which gives us some insight about what actually happened in the guest following the kernel

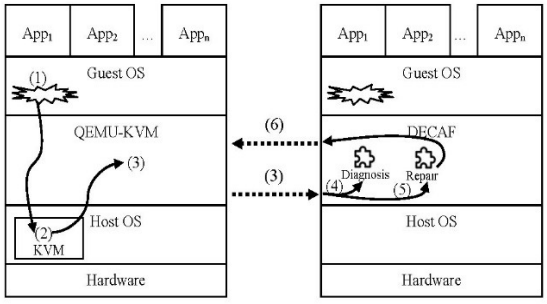

Figure 6. Overall Design of Kernel Panic Triage Using Live Migration Bafsed on-the-fly Software Emulation

panic. However, instruction trace analysis alone is not sufficient because it does not give information about kernel data (e.g., the content of a buffer that is passed as input to *printk*). In order to retrieve such kernel data, we also developed a new DECAF plugin.

5.3.2.1 Recover the Control Flow

To locate the source of the kernel panic, we first save the state of the virtual machine and then load the *tracecap* plugin in DECAF to obtain an instruction trace of the guest kernel. Once setup, we then allow the virtual machine to continue execution so as to obtain the instruction trace. Based on the instruction trace, and combining it with kernel symbol information (e.g., /proc/kallsyms and system.map), we conclude the guest kernel encountered a kernel oops, and the kernel is trying to handle it as depicted in Figure 5.

5.3.2.2 Recover the Panic Message

Although the recovered kernel control flow confirms a kernel panic in the guest, it does not give further information about the cause of the kernel panic, e.g., which kernel component triggers the null pointer dereference, what the offending instruction is, and where this instruction is located. From the source code of the guest kernel, the function *show_fault_oops* would print "Unable to handle kernel NULL pointer ..." and the function *__die* would print contextual information of the kernel panic, including stack frame and value of the registers. Unfortunately, such useful information is not visible on the monitor of the guest VM before the guest OS hangs indefinitely. It should be written into /var/log/syslog of the guest, but since the guest hangs, it is not flushed to the guest's hard disk, which means that once the guest is rebooted, such information cannot be found in /var/log/syslog.

Therefore, we utilize the DECAF support for guest function hooking to retrieve such information. Both *show_fault_oops* and *__die* above invoke *printk* to dump the messages. Therefore, we can hook *printk*, which means that we will intercept the guest kernel execution at the entrance of *printk*, and then we can access information that is passed to *printk* as input parameters, such as the string “Unable to handle kernel NULL pointer ...”

However, directly hooking *printk* has one practical issue: the hook handler has to parse the input to *printk*, which is a non-trivial job (e.g., to interpret the semantic of format strings like "%x"). Therefore, we look into the source code of *printk* and find a function *cont_add* that is invoked by *printk* and handles already formatted output. Hooking *cont_add* can directly give us a pointer to the *printk* output buffer with already formatted message, which is much more convenient than hooking *printk*. Specifically, the type definition of *cont_add* is:

```
static bool cont_add (int facility, int level, const char *text, size_t len)
```

wherein *text* points to the formatted output buffer, and *len* is the number of bytes in the buffer.

Based on the above analysis, we develop a DECAF plugin that hooks *cont_add* and retrieved the error messages from the guest memory and logs it to the user. This entire process illustrates the advantages of using emulation for in depth analysis.

5.3.3 Recovery from Guest Kernel Error in DECAF

A second thing that an analyst can do after receiving a guest VM with a kernel panic is to recover the guest kernel from the error before resuming its execution. In our case, the initial saved state from Section 5.3.2.1 is now restored for triage. In other words, we saved the state at the beginning of analysis so we could learn more about the problem at hand by allowing the virtual machine to continue execution until it hangs. We then devised a resolution strategy based on what we have learned and restored the machine to the previous state so that we can remediate the issue.

In the case of a buggy kernel module, the goal is to undo all the impact that the module has to the rest of the system, e.g., cleanly unload the kernel module. An analyst may try two options: the first one is to abort the faulting function (i.e., causing it to return immediately to the caller, as if the faulty instruction is *return*) and leave to the user to unload the buggy module; and the second option is to abort the faulting function and then force the kernel execution to a path that unloads the buggy kernel module.

5.3.3.1 Aborting the Execution of a Faulting Function

Since a kernel panic occurs in the middle of execution, a function is partially executed. A partial execution of a function can have two kinds of impact: global impact and local impact. Global impact is the change to things like global variables, the heap, and the file system, and local impact includes change to CPU registers and the stack.

Therefore, the impact analysis of a partial function execution would process a sequence of instructions between the entrance of the function to the faulting instruction, and for each instruction in the sequence, it will calculate the instruction's impact, whether global or local. Then a recovery plan would consist of operations that undo the impact such as reverse execution. For example, the undo operation for a *malloc* function call is *free*, and the undo operation for a *push* instruction is a *pop* instruction. In general, not every instruction can be undone. For example, a modification to a global variable cannot be undone unless the original value of the global variable is known. Therefore, whether the impact of a partial function execution can be undone is case specific.

For demonstration purposes, we ensured that our buggy module represents a case that can be undone. The reason is that the buggy function has no global impact before it performs the null pointer dereference, and it does not invoke any other function that may have side effects.

5.3.3.2 System Recovery

To recover the system, we first obtained a backwards instruction trace by using the debug information obtained from *cont_add* as a guide to where the current function resides in memory. We then proceeded to reverse execute each instruction in the offending function so as to restore the state of the virtual machine to before the function with the null pointer exception is called. We also patched the offending function so that it returns immediately thereby preventing future kernel panics. Once the exception state has been cleaned up, the virtual machine is live migrated back to KVM for continued execution.

As an additional test, we also performed an experiment where execution resumes at the beginning of a kernel module removal routine. We implemented such a routine by creating a variant of an existing kernel function *sys_delete_module*. The main difference between our implementation and that of the original function is that our routine does not take parameters from the user space (*sys_delete_module* does because it is a system call

handler); moreover, it schedules a delayed work item that will carry out the main operations of *sys_delete_module* at a later time. This is needed because kernel panics are handled in an "interrupt handling context" where code executions must be brief so further interrupts are not blocked. Removing the offending module should be executed in a normal process context instead, which is why a delayed work item is used. (Currently, the new function is implemented as a kernel module that has to be loaded in the guest VM).

A video demonstration of this can be seen at: http://www.cs.fiu.edu/~weijp/MOSE/demo1.html.

6. CASE STUDY TWO: PROBING AND PATCHING OF SECURITY VULNERABILITIES

6.1 Motivation

Latent vulnerabilities are constantly discovered in complex and mission-critical systems. For example, the SSL Heartbleed bug disclosed in 2014 called for immediate patching of this vulnerability in many mission-critical systems (e.g., SCADA systems). However, patching of such vulnerabilities is challenging. First, one has to confirm the existence of such bugs by probing the system, but probing on the spot is expensive, e.g., we need instruction-level taint analysis to understand whether malicious input can reach the *malloc* size variable of an SSL server [1] and if sensitive data in memory reaches the network through the buffer over read. Second, probing on the spot is risky, e.g., anecdotal evidence has shown that it may cause a SCADA control system to crash. Lastly, guaranteeing the correctness of the patch (e.g., it will not trigger other vulnerabilities) is hard.

6.2 Probing and patching of an Heartbleed bug using live migration based on-the-fly emulation

Leveraging the prototype described in Section 4, we built a simple use case. We first run a buggy OpenSSL server in a VM on top of QEMU/KVM that only has our heterogeneous migration patches. This is used to represent a normal user who does not have any specialized analysis capabilities built into the hypervisor. Next, we migrate the guest VM into DECAF and verify that the OpenSSL server continues to run. Then we load the *tracecap* plugin into DECAF to obtain an instruction trace of the OpenSSL process. Using this instruction trace and virtual machine introspection (built into DECAF), we locate the Heartbleed bug in the OpenSSL server and implement a binary patch. Then we implemented a binary patching plugin for DECAF that uses DECAF's virtual machine introspection support to locate the library of interest (e.g. libssl) and then live patch the library instructions in memory. Finally, we migrate the OpanSSL server back to QEMU/KVM and confirm that the Heartbleed vulnerability has been eliminated. A video demonstration of this can be seen at: http://www.cs.fiu.edu/~weijp/MOSE/demo2.html.

7. CASE STUDY THREE: ON-DEMAND SYMBOLIC EXECUTION

7.1 Motivation

DSLab's S2E is a platform for analyzing the properties and behavior of software systems [2], and one of its key capabilities is selective symbolic execution. Since this capability is built on top of QEMU, it inherits the slowness of the dynamic binary translation that QEMU normally uses. To improve user experience especially when large guest VMs are analyzed, S2E introduces an experimental feature that supports taking snapshots in KVM and later resuming them in QEMU to carry out symbolic execution [14]. However, the switching from KVM to S2E is still inconvenient: a user is expected to know when a guest VM is ready for symbolic execution and manually take the snapshot beforehand, which is problematic when the symbolic execution starts in the middle of a running application because the user may not be able to suspend the VM right before the symbolic execution starts. Therefore, we propose an enhancement to S2E that detects the need for symbolic execution in KVM and automatically live migrate the guest VM to S2E for continued symbolic execution.

7.2 Design and Implementation

S2E introduces custom instructions as the programming interface between applications and the S2E platform: when such instructions are emulated by the S2E execution engine, they are interpreted as requests for symbolic execution. On non-S2E systems, however, such instructions trigger invalid instruction exceptions and cause the applications to crash.

We note that the exceptions caused by S2E custom instructions are a good hint that the application needs symbolic execution. Therefore, we extend the KVM kernel module to capture such exceptions and initiate a live migration from QEMU/KVM to S2E. The overall design is similar to Figure 6 except for some technical details. For example, we check invalid opcode exceptions instead of page fault exceptions, and when an invalid opcode exception is encountered we further verify that it is indeed caused by an S2E custom instruction (i.e., the opcode of the faulting instruction is `0x0f,0x3f`), by inspecting the guest emulation context. When a S2E instruction is confirmed, we pass control back to QEMU, setting the exit reason as KVM_EXIT_INTERNAL_ERROR and the sub-error code as 5. We also extended QEMU/KVM's execution engine accordingly to initiate a live migration to a S2E platform.

We have confirmed the feasibility of our approach in two experiments. In the first experiment, we ran a simple program (on a Windows Virtual Machine) that is instrumented in the source code with S2E instructions to inject symbolic values (e.g., by calling *s2e_make_symbolic*). In the second experiment, we use the S2E *init_env* library and the *s2ecmd* tool to symbolically (or concolically) execute an existing Linux program *echo*. In both cases, we were able to automatically migrate the applications (along with the guest VM) from QEMU/KVM to S2E and then begin symbolic analysis in S2E. A video demonstration of this as well as the source code can be found at: http://www.cs.fiu.edu/~weijp/MOSE/demo3.html.

8. PERFORMANCE EVALUATION

8.1 Experimental Setup

For experimental setup we ran QEMU/KVM and DECAF on two different computers: QEMU/KVM was on an Intel Core 2 Duo running at 2.5 GHz with 1GB of memory, and DECAF was on an Intel Core 2 Duo running at 3.0 GHz with 4GB of memory. Each computer has an Intel PRO/1000 PCI Express network interface card, connected to a common switch. The virtual machine image file was stored in a CIFS (Common Internet File System) network share mounted in both machines. Therefore, during migration we

only had to transfer the states of the VM, not the entire image file. We installed an Apache web server in the virtual machine, and we configured the virtual machine to use DHCP so that during migration its IP address does not change, which ensures that after migration the web client can reach the Apache server at the same IP address, so it does not have to be restarted.

We adapted a web server benchmark tool 'weighttp' to measure the liveness of the Apache server, by continuously downloading a 1-KB text file from the server and reporting the throughput (in KByte/s) every 30 milliseconds. The intuition is that a non-zero measurement of the throughput indicates that the server is available, while a throughout of zero indicates that the server is not available due to migration or server crash. Our main focus was to show that our proposed method can recover from server crash and take care of existing client connections after the recovery.

8.2 Downtime during a Normal Migration

We began our evaluation by measuring the downtime of the Apache web server during a normal migration, which indicates how fast a virtual machine can be migrated between QEMU/KVM and DECAF that run on different hosts. As Figure 7(a) shows, the throughput dropped a little when we initiated a migration around the 30th second in QEMU/KVM, but it became zero only for about 300 ms around the 33rd second and it quickly ramped up after the Apache server is resumed in DECAF. The migration back to QEMU/KVM shows a similar pattern in terms of the throughput.

8.3 Downtime during a Kernel Panic

Next we evaluated the effectiveness of our kernel panic recovery on the Apache web service. Figure 7(b) shows the throughput measurement without our kernel panic recovery. We can see that the throughput dropped to zero after the 30th second, when the kernel panic was triggered in the virtual machine running on top of QEMU/KVM. Figure 7(c) shows the throughput measurement with our kernel panic recovery. As we can see, the throughput dropped to zero around the 30th second, when the kernel panic was triggered in QEMU/KVM. However, the throughput started to increase around the 33rd second because our recovery tool had automatically migrated the crashed VM to DECAF, recovered it from the crash, and resumed its execution, including that of the Apache server. Around the 45th second, we migrated the recovered VM back to QEMU/KVM. The comparison between Figure 7(b) and Figure 7(c) clearly demonstrates the impact of our kernel panic recovery on applications like Apache.

From Figure 7(a) and Figure 7(c), we can see some interesting difference surrounding the migration from QEMU/KVM to DECAF (i.e., the time window marked “M1”) with respect to the Apache's server's throughput. First, in a normal migration, the Apache throughput fluctuated below its peak level and dropped to zero in 3 seconds, but during a kernel panic recovery, its throughput dropped to zero right away. This is because during a normal migration, the guest VM's state is first transmitted to the destination machine before its execution is fully suspended, which means that the Apache server kept servicing the client during the VM state transmission but the network traffic due to the VM states reduced the available bandwidth to Apache, resulting in dropped throughput. On the other hand, when a kernel panic was triggered in Figure 7(c), the Apache server immediately stopped execution, which caused its throughput to drop to zero. The second difference is that the downtime of the Apache server is longer during a kernel crash recovery (about 3 seconds) than in a normal case (about 300 ms). This is because during the crash recovery, the Apache server is suspended during the entire migration, whereas it can run at a reduced capacity during a normal migration.

From Figure 7(a) and Figure 7(c) we can also note that the Apache server has a higher throughput in DECAF than in QEMU/KVM. To verify whether this is true in general, we ran *netcat* [18] in the guest VM to compare the network throughput offered by DECAF and QEMU/KVM on the same host (Intel Core 2 Duo running at 3.0 GHz with 4GB of memory), and we got a send rate of 17.8 Mbps for DECAF and 4.4 Mbps for QEMU/KVM, and a receive rate of 27.1 Mbps for DECAF and 7.1 Mbps for QEMU/KVM. We believe that the lower network performance of QEMU/KVM is caused by a large number of VM Exits due to network device emulation. VM Exists trigger expensive context switches between QEMU/KVM and the host kernel. For example, when *netcat* was run to receive data, the rate of VM Exit increased by 34 times (from 725 per second to 25,344 per second), and when *netcat* was run to send data, the rate of VM Exit increased by 268 times (from 725 per second to 194,725 per second). Comparatively, pinging an external host from inside the guest VM incurred only a 2.2 fold increase in VM Exit rate (from 725 per second to 1,613 per second). Therefore, there seems to be a strong correlation between the guest VM’s network activities and the VM Exit events, and we know that VM Exits incur performance overhead.

9. FUTURE DIRECTIONS

The successful development of a prototype for live migration based demand emulation opens the ground for many useful applications. We can apply our architecture to the following usage scenarios:

Scenario 1: Analysis of exploits against a server. When we detect the abnormal behavior of a server, it is important to understand whether some kind of exploit has happened, especially when a zero-day vulnerability is exploited. Using our architecture, we can take periodic snapshots and record network traffic (e.g., client requests) in QEMU/KVM; when an anomaly is detected, we can revert the server to a previous snapshot, live migrate the server to DECAF, and replay the network traffic since that snapshot to analyze the server’s execution. By doing this, we can perform detailed analysis of real exploitations.

Scenario 2: Analysis of emulation-resistant malware. Advanced malware (e.g., botnets) is capable of detecting whether it is running inside an emulator and if so changing its behavior to evade detection. Using our architecture, we can live-migrate the clone of a potentially compromised server to DECAF, but let the original server continue to run in parallel with its clone. In the meantime, we can record the network traffic in the original server and use it to guide the emulated execution of the server clone in DECAF. Since malware in QEMU/KVM is likely to reveal its real behavior, the recorded network traffic can contain useful information about malware’s logic (e.g., the command and control message from a bot master). By replaying such network traffic in DECAF, we can trigger the malicious logic in malware and then be able to analyze its behavior.

Another future work is migration across WAN (wide area network). We have performed experiments with live-migration across WANs. While feasible, the total migration took over one

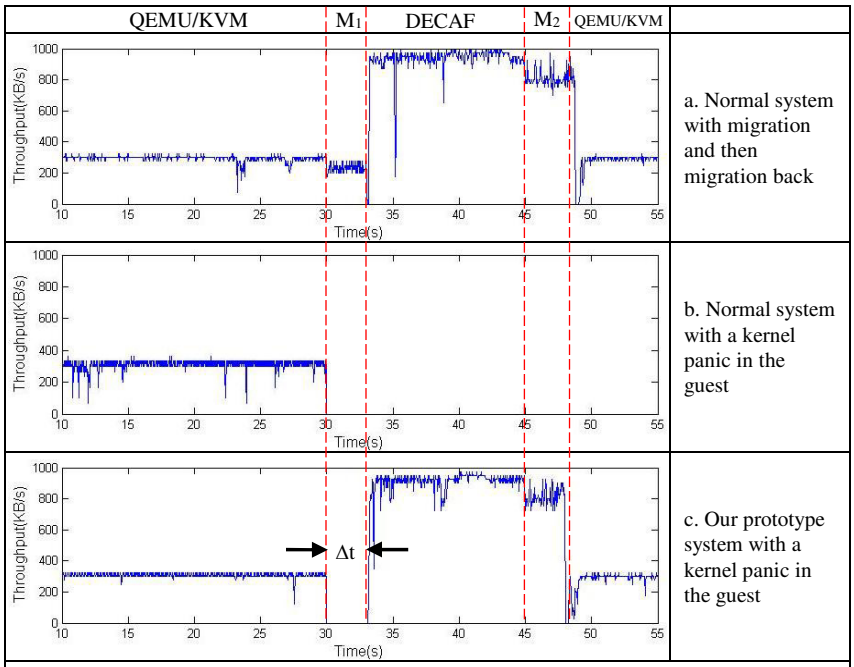

Figure 7: Throughput of an Apache Server in Three Scenarios (Δt is the Time to Triage)

hour of time. How to improve the performance is an area of future exploration.

10. CONCLUSION

We have proposed the idea of live migration based on-the-fly software emulation, which significantly advances the state of the art: it is analysis-flexible, supports closed source OSes, and uses modern hardware virtualization extensions. We have built a prototype based on QEMU/KVM and DECAF. Based on the prototype, we construct three use cases (kernel panic triage, patching of security vulnerabilities, and on-demand symbolic execution) to demonstrate the benefits of our idea. We also compare the performance (throughput and downtime) of a web server on top of our prototype and with/without the kernel panic triage, which shows sub-second interruption of service during live migration.

11. ACKNOWLEDGEMENTS

This work was supported by the United States Air Force Research Laboratory (AFRL) Visiting Faculty Research Program (VFRP) and its extension grant. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the United States Air Force Research Laboratory.

12. REFERENCES

[1] Sean Cassidy. Diagnosis of the OpenSSL Heartbleed Bug. http://www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html. Accessed May 17, 2015.

[2] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems, February 2012.

[3] Chow, J., Garfinkel, T., & Chen, P. Decoupling dynamic program analysis from execution in virtual environments. USENIX 2008 Annual Technical Conference, pp. 1–14.

[4] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, R. Whelan. Repeatable Reverse Engineering for the Greater Good with PANDA. Columbia University Technical Report, CUCS-023-14, October, 2014.

[5] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. Proceedings of ISSTA'14.

[6] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. Proceedings of EuroSys '06.

[7] Intel Corporation. Intel 64 and IA-32 Architectures: Software Developer's Manual, Volume 3A.

[8] Kernel-based Virtual Machine. http://www.linux-kvm.org/page/Main_Page

[9] Live migration in KVM. http://www.linux-kvm.org/page/Migration

[10] Microsoft: Step by Step Guide for live migration. http://technet.microsoft.com/en-us/library/dd446679.aspx

[11] OpenVZ checkpointing and live migration. http://wiki.openvz.org/Checkpointing_and_live_migration

[12] QEMU. http://wiki.qemu.org/Main_Page

[13] Swift, M. M., Annamalai, M., Bershad, B. N., & Levy, H. M. (2006). Recovering device drivers. ACM Transactions on Computer Systems, 24(4), 333–360.

[14] s2e Team. Experimental KVM Snapshot Support. https://github.com/dslab-epfl/s2e/blob/master/docs/ImageInstallation.rst#experimental-kvm-snapshot-support

[15] VMware vMotion: Virtual Machine Live Migration. http://www.vmware.com/products/vsphere/features/vmotion

[16] HOWTO Article about Xen migration. http://www.linux.com/archive/feature/55773

[17] Yan, L., Jayachandra, M., Zhang, M., & Yin, H. (2012). V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. VEE (pp. 227–237).

[18] netcat – Linux man page, at http://linux.die.net/man/1/nc.