

# SANDIA REPORT

SAND2016-9616

Unlimited Release

Printed September, 2016

## Staghorn: An Automated Large-Scale Distributed System Analysis Platform

Kasimir Gabert, Ian Burns, Steven Elliott, Jenna Kallaher, Adam Vail

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Staghorn: An Automated Large-Scale Distributed System Analysis Platform

Kasimir Gabert (5638), Ian Burns (9526), Steven Elliott (5634)  
Jenna Kallaher (5631), Adam Vail (5634)

Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, New Mexico 87185-0620

## Abstract

Conducting experiments on large-scale distributed computing systems is becoming significantly easier with the assistance of emulation. Researchers can now create a model of a distributed computing environment and then generate a virtual, laboratory copy of the entire system composed of potentially thousands of virtual machines, switches, and software. The use of real software, running at clock rate in full virtual machines, allows experiments to produce meaningful results without necessitating a full understanding of all model components. However, the ability to inspect and modify elements within these models is bound by the limitation that such modifications must compete with the model, either running in or alongside it. This inhibits entire classes of analyses from being conducted upon these models. We developed a mechanism to snapshot an entire emulation-based model as it is running. This allows us to “freeze time” and subsequently fork execution, replay execution, modify arbitrary parts of the model, or deeply explore the model. This snapshot includes capturing packets in transit and other input/output state along with the running virtual machines. We were able to build this system in Linux using Open vSwitch and Kernel Virtual Machines on top of Sandia’s emulation platform Firewheel. This primitive opens the door to numerous subsequent analyses on models, including state space exploration, debugging distributed systems, performance optimizations, improved training environments, and improved experiment repeatability.

# Acknowledgments

Zach Benz	Critical Systems Security	5624
David Burton	Embedded Systems Engineering	5632
Justin Ford	Embedded Systems Engineering II	5634
Todd Jones	Cyber Initiatives	5638
Shelly Leger	Computer Science Dept I	5636
John Rowe	Defense Systems & Assessments	5000
Tan Thai	Information Systems Analysis Center	5600

# Contents

<b>Nomenclature</b>	<b>11</b>
<b>1 Introduction</b>	<b>13</b>
1.1 Related Work	14
1.2 Use Cases	15
1.2.1 Vulnerability Analysis	15
1.2.2 Debugging Distributed Systems	16
1.2.3 Debugging Emulation-based Experiments	16
1.2.4 Experimental Repeatability	17
Regression Testing	17
Experimental Non-determinism	17
1.2.5 Training Environments	18
<b>2 Design</b>	<b>19</b>
2.1 Full System Snapshot Staging	19
2.1.1 NTP vs PTP Time Synchronization	19
2.1.2 Fully-Distributed Snapshot Staging Architecture	20
2.2 Virtual Machine Time Perception	21
2.3 Virtual Machine State Snapshots	24
2.3.1 QEMU Migration Snapshots	24
QCOW2 Disks	24
LVM Disks	24
2.4 Snapshotting Networks	25

2.4.1	Design Considerations .....	25
	Packet Latency and Ordering .....	25
	Using Netlink .....	26
	Staghorn Modification Placement .....	27
<b>3</b>	<b>Implementation</b>	<b>31</b>
3.1	Full System Snapshot Staging .....	31
3.1.1	Staghorn Triggers .....	32
3.2	Virtual Machine Snapshots .....	32
3.3	Network Snapshots .....	33
3.4	netsnap_user program .....	34
3.5	Staghorn Packet File Format .....	35
3.6	Installation .....	36
3.6.1	Integration with Firewheel .....	36
<b>4</b>	<b>Evaluation</b>	<b>37</b>
4.1	Benchmarking Studies .....	37
4.1.1	Precisetimer test .....	37
4.1.2	RabbitMQ Message Speed .....	38
4.1.3	Snapshot timing test .....	38
4.1.4	Dropped Packets .....	39
4.1.5	Guest VM Agnostic .....	39
4.2	Use Cases Tested .....	41
4.2.1	Vulnerability Analysis .....	41
4.2.2	Debugging Distributed Systems .....	43
	Remote Debugging .....	43
4.2.3	Debugging Emulated Experiments .....	44

<b>5 Conclusion</b>	<b>45</b>
<b>References</b>	<b>46</b>
 <b>Appendix</b>	
<b>A Packet Spacing</b>	<b>55</b>
A.1 Timestamps in the Linux Kernel . . . . .	55
A.2 Delays and Timing . . . . .	57
<b>B VM State Snapshot Optimization</b>	<b>59</b>
B.1 Introduction . . . . .	59
B.2 Process-Level Snapshots . . . . .	59
B.3 QEMU Fork-based Snapshots . . . . .	68
B.4 KVM IOCTLs Traces . . . . .	70
B.5 Replaying KVM IOCTLs . . . . .	72

# List of Figures

1.1	Example of how Staghorn can optimize common prefix regression testing. If each state and test costs one operation, without taking any snapshots there will be 19 operations performed. If Staghorn is used to take a snapshot at every state, the number of operations is reduced to 9. In this example any state has a cost of one, however in many real-world regression tests simply getting to a state may take hours. . . . .	17
2.1	An overview of the Staghorn snapshot staging and messaging system. . . . .	21
2.2	Timing tests measuring whether a virtual machine can detect a difference in its time due to being snapshotted. In these figures, “with” indicates that snapshots are being performed on the VM and “without” indicates that the VM is running without any snapshots being performed. . . . .	23
2.3	Example of the <i>OWD</i> introduced by Staghorn . . . . .	27
2.4	Rough outline of a packet’s progression to OVS [65]. The Staghorn entry and exit points are highlighted. . . . .	29
4.1	The results of the precisetimer test. Each iteration had a sleep target of one second into the future. The mean error was 28.05 nanoseconds. . . . .	38
4.2	The results of the RabbitMQ introduced latency error test. The mean error was 1.26 milliseconds. The ribbon is a 95% confidence interval from the smoothing line. This test was performed both through RabbitMQ on localhost and on two machines within the testing cluster. . . . .	39
4.3	The results of measuring how long it takes quiesce a single CPU. As density and/or loading increases, each VM’s CPUs will be quiesced in parallel but load and density is a limiting factor. This test was performed on a node with 16 cores, so any density higher than 15 (one core available to perform the snapshot) would not be recommended. This test contains measurement error as we are not capturing the moment when the CPUs are stopped but instead once we can query, out-of-band, that they are stopped. . . . .	40
4.4	Staghorn can easily fuzz packets by triggering a snapshot on a target packet, modifying the packet, continuing the experiment, and returning to the pre-modification state . . . . .	42



4.5	Setting a breakpoint in JDB.....	44
4.6	The breakpoint being hit in JDB causing a snapshot to be triggered. ....	44

# List of Tables

3.1	The interface functions for staging actions. ....	31
3.2	The actions used to snapshot VM state using QEMU's migration-based snapshots. ....	33
3.3	The actions used to snapshot network state. ....	34
3.4	The commands used in <code>netsnap_user</code> . ....	35

# Nomenclature

**VM** Virtual machine

**OVS** Open vSwitch, a flexible software switch used in Linux

**KVM** Kernel-based Virtual Machine, a virtualization extension for Linux

**QEMU** Quick Emulator, a hypervisor commonly used in conjunction with KVM to run virtual machines on Linux

**QMP** QEMU Machine Protocol

**QCOW2** QEMU Copy On Write 2, a file format used by QEMU

**COW** Copy on write

**LVM** Logical Volume Manager, a Linux management system for logical volumes

**NTP** Network Time Protocol

**PTP** Precision Time Protocol

**ICMP** Internet Control Message Protocol

**IP** Internet Protocol

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

**VXLAN** Virtual Extensible LAN

**GRE** Generic Routing Encapsulation

**CPU** Central processing unit

**THP** Transparent Huge Page

**VA** Vulnerability Analysis

**TSC** Time Stamp Counter

**ACPI** Advanced Configuration and Power Interface

**LAN** Local Area Network

**I/O** Input/Output of a computer to other devices around it



# Chapter 1

## Introduction

The last decade has seen a tremendous increase in scale, complexity, and use of distributed systems. Nearly every network-enabled service is being redesigned into a loosely-coupled distributed system, from video services, to email servers, to private cloud infrastructure. This shift necessitates the capability to thoroughly evaluate the full distributed system.

Full-scale emulations are a useful tool to perform such evaluation. By leveraging an emulation, users are able to build virtual, laboratory copies of distributed computing and control systems using virtualization techniques. These models may include tens of thousands of heterogeneous virtual machines running numerous software applications and connected with thousands of virtual networks. These models are built in a way that the virtual machines will begin as vanilla machines and configure themselves programmatically after the experiment begins. As a result, they are currently used as the basis for controlled experiments to analyze system behaviors. Currently, entire classes of analyses are unable to be conducted at scale upon emulated distributed systems because no available mechanism captures the entire state of the system. As a result, users face challenges including: large non-determinism impacting experiment outcomes, long distributed system debug cycles, inefficient or nonexistent distributed system vulnerability analysis capabilities, and inefficient system-wide regression tests.

We present Staghorn, a platform that introduces a novel capability to perform full system snapshots of distributed system emulations. Full system snapshots include all of the virtual machines and their state, the disk activity by the VMs, and the virtual networks including packets in transit. This powerful primitive allows a model to be studied in a variety of new ways. We have demonstrated new vulnerability analysis capabilities which will enable researchers to discover, study, and fix software bugs in large distributed applications. Staghorn dramatically improves debugging such applications by halting all execution almost instantly when a breakpoint is hit. Researchers using Staghorn are now able to examine non-determinism of distributed systems which can lead to advances in understanding emulated environments and system complexity. When running regression tests software quality engineers can use Staghorn to save common execution prefixes used by multiple test cases, dramatically reducing runtime. Staghorn can be leveraged for training network defenders by enabling participants to repeatedly attempt difficult objectives without resetting the entire system. We show that Staghorn enables users to conduct analysis on distributed systems at scales that are an order of magnitude larger than any previously published studies.

Staghorn has been implemented on top of Linux/QEMU/KVM/OVS, and was developed as an extension of Firewheel [46], a Sandia developed emulation testbed that can instantiate environments of tens of thousands of virtual machines across clusters of physical machines.

Generally, our emulation-based models by design run at processor clock speed. This has many implications, most significant of which is that these models cannot be handled like a traditional computer simulation in which checkpoints can be made and the entire simulation can be processed offline, or without impacting the model in any way. Staghorn is the first step in opening up this larger world of offline processing to large-scale emulation-based models by allowing such full offline inspections and modifications to occur once a snapshot has been taken.

Our key contributions are:

- A full-system snapshot and restore capability for Sandia’s large-scale emulation-based model environments which preserves network and I/O state
- A network modification system that allows for modification of Ethernet frame contents and delivery, or the introduction and removal of frames, during a snapshot
- The evaluation of this capability on real-world use-cases

This involves stopping thousands of virtual machines across hundreds of physical computers in a very tight time window. To accomplish this, we built an orchestration system to stage the snapshots, techniques to snapshot the virtual machines, and a new network snapshot and monitoring system. We evaluated this capability on real-world distributed systems. We additionally developed a new process-level snapshot technique for arbitrary Linux processes.

This report is broken up into five chapters. Chapter 1 contains the introduction, related work, and five highlighted potential use cases. Chapter 2 covers the design considerations for each of the Staghorn components. Chapter 3 describes many of the implementation details for Staghorn. Chapter 4 contains several benchmarks for Staghorn and demonstrates several of the previously outlined use cases. Finally, Chapter 5 concludes. The appendices contain details regarding advanced techniques that we pursued.

## 1.1 Related Work

As distributed systems have become ubiquitous, extensive research has occurred in distributed system analysis. All of this work has roots in more traditional software analysis. The existing work can be roughly broken into three categories: model checking, symbolic execution, and dynamic analysis. None of the existing work in these categories can support a meaningful VA in large-scale environments. Current state-of-the-art platforms, such as

Turret-W [53], only run on one physical machine. Firewheel has been shown to scale to 96 physical machines and can likely scale higher.

Model checking [70, 64, 100, 88] is used to demonstrate correctness of a system, but suffers when used in VA as it can only identify illegal states that are reached through a legal sequence in the model. Models typically only have protocol-legal sequences to remain tractable. Staghorn is not limited by strictly legal protocol sequences and instead execution will be determined by a greedy search. CrystalBall [100] works around this problem by deploying a model checker on each deployed node of the system and continuously exploring the state space within the node’s neighborhood as real-world packets enter the system. Using this technique CrystalBall is able to inform a deployed system of potential error states that it may get into. While CrystalBall may detect real-time attacks, it won’t discover illegal attack paths and therefore is not suitable as a VA platform.

Symbolic execution [81, 87, 82] is used to find constraints on inputs that will lead to illegal states. These systems provide symbolic inputs to the distributed systems and execution of conditional branches may induce constraints. An input can then be generated that will satisfy the given constraints and result in the distributed system reaching an error state. While symbolic execution is more robust than model checking, as it can run algorithms directly, it is not feasible to keep symbolic states for a complex, large-scale distributed system. Instead, Staghorn evaluates a metric (for example a performance metric, proximity to an error state, or stability and imbalance in the system) across the system to identify desired states.

Dynamic analysis [62, 53, 58] attempts to overcome the above problems by running the distributed system natively. Several projects [62, 53] have worked towards building single-host analysis platforms that are capable of finding attack paths in small distributed systems. At the moment, there are no other known cluster-wide dynamic analysis platforms due to the challenges with implementing such systems. Building off of Sandia’s cyber analysis platforms and pulling together work from distributed snapshots, virtual clocks, message modification systems, and virtualization technologies, Staghorn fills this gap.

## 1.2 Use Cases

### 1.2.1 Vulnerability Analysis

Vulnerability analysis (VA) is a critical part of developing secure software that assists in software verification and increases confidence that the produced system behaves as designed [9]. As part of this security verification, fuzz testing, or attempting to find uncaught error conditions by providing deliberately invalid data to an application, is often used [9, 89]. In most cases, fuzzing occurs in a stand alone context, for example the binary interface to a dynamic library. In network applications, it occurs by corrupting packets, reordering packets, injecting new packets, or dropping packets and keying on incorrect handling through some

monitoring software. However, dumb fuzzers, or fuzzers that do not maintain any knowledge about a particular protocol, are not efficient for properly evaluating stateful applications [22]. Stateful network protocol fuzzers, such as Peach [13], allow users to outline a protocols state machine so that each state can be appropriately stressed. In these cases an end user still needs to generate the state diagram of a complex distributed protocol which may vary by a particular applications implementation. AutoFuzz [49] attempts to generate the protocol's state machine but uses a proxy server to man-in-the-middle traffic. Additionally, neither AutoFuzz nor other stateful fuzzers provide a user with the ability to reset the entire system to a given state. Staghorn can drastically improve distributed system VA by snapshotting the entire system's initial state which guarantees state consistency between test cases. While Staghorn has not been integrated with any existing fuzzers, we demonstrate that Staghorn can even enhance dumb fuzzers.

## 1.2.2 Debugging Distributed Systems

Debugging distributed systems is difficult due to sheer complexity of the component interactions. This is further complicated by the independence of the system parts. Suppose *Component<sub>B</sub>* is responsible for processing data sent by *Component<sub>A</sub>*. If an error occurs in *Component<sub>B</sub>*, component *Component<sub>A</sub>* will continue to send data, even though component *Component<sub>B</sub>* can no longer process it. In some cases, the continuation of the system may even exacerbate the failure or hide the root cause. An ideal solution to this problem is to pause the entire system while diagnosing the error, allowing the developer to determine the root cause without further crashes.

## 1.2.3 Debugging Emulation-based Experiments

Discovering and resolving experimental problems is a time consuming process. This is especially true for long-running experiments in which a bug may occur hours or even days into the experiment. There exist several debuggers for high-performance computing applications that are designed to handle this, but for many distributed systems this is an open problem [96, 91, 97]. Additionally, it may take several attempts to fully fix the bug, each of which requires another multi-day long test. Staghorn can drastically improve this tedious cycle by allowing users to revert to a state before the error occurred. Users can automate Staghorn to take snapshots every five minutes. Then if an error occurs the user can revert to the closest snapshot, troubleshoot the issue, and test the fix repeatably. By using these interval snapshots, Staghorn has reduced the debug cycle from hours or days to just minutes. Users could also choose to set a time-based trigger for deterministic bugs. Many experiments could take hours to correctly set up the system, for example an experiment involving Microsoft Exchange requires several hours of automated setup. Using Staghorn, we were able to save a researcher hours while debugging an emulation model by snapshotting it before the error.



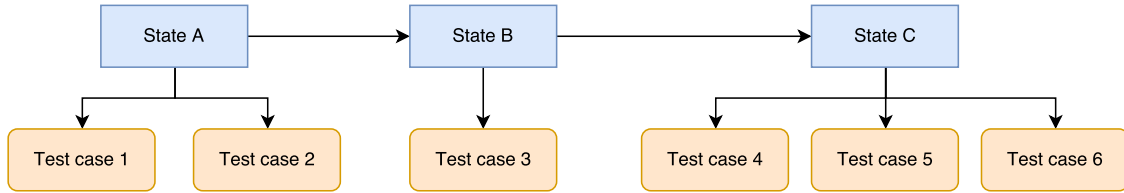


Figure 1.1: Example of how Staghorn can optimize common prefix regression testing. If each state and test costs one operation, without taking any snapshots there will be 19 operations performed. If Staghorn is used to take a snapshot at every state, the number of operations is reduced to 9. In this example any state has a cost of one, however in many real-world regression tests simply getting to a state may take hours.

## 1.2.4 Experimental Repeatability

This section outlines how Staghorn can be leveraged to optimize the time required to run distributed system experiments. Furthermore, we give examples of how Staghorn can assist in determining and even reducing the non-determinism of the system.

### Regression Testing

Each small change to a distributed application requires regression tests. However, depending on the test resources available, there may be many changes but only a single test setup. Furthermore, tests can take many hours to complete causing delays in applying patches. When running tests on distributed systems, each test case requires the system to be in a particular state leading to lots of overhead. With Staghorn, common state prefixes can be saved so that multiple test cases can be run using the same state. Figure 1.1 demonstrates using Staghorn to run multiple test cases from each state. Staghorn not only improves the speed at which regression tests can occur, it also reduces the non-determinism between test cases requiring the same state, thereby helping to address two of the toughest challenges facing distributed system regression testing [83].

### Experimental Non-determinism

Staghorn can also be used as a tool for understanding experimental non-determinism. For example, starting from a single state, a user can play the system forward for 10 seconds. By repeating this process thousands of times, the researcher can then find the difference between all of the snapshots to infer the non-determinism during that time frame. By leveraging Staghorn we can begin to understand the non-determinism of distributed system experiments and discover their impact on experimental results.

## 1.2.5 Training Environments

Cyber defense training environments can be viewed as an experiment in which there is a human-in-the-loop. Emulation with Firewheel enables a repeatable training environment in which a human defender can attempt to protect a system against automated attacks. One potential training situation may evaluate how well a defender can harden a system against different attacks. In a typical emulation, a defender either has to set up the hardened system many times (either manually or through automation) or generate a pre-baked hardened VM. Neither of these situations is ideal as the first option wastes valuable training time and the latter creates a complicated and large VM library. Staghorn can be leveraged to snapshot the entire system once it is setup. Then many different attacks can be used against the same initial setup to evaluate how well the defender did. These snapshots have the added benefit of being able to examine and replay attacks which succeeded enabling the defender to learn how to correct their mistakes. Another training scenario is when the user needs to actively defend the network. In this case, Staghorn is the perfect tool to create checkpoints during the exercise so that the defender can revert to a previous checkpoint if an objective is failed. This enables a video game-like environment where users can practice defending against difficult attacks multiple times.

# Chapter 2

## Design

In this chapter we describe the design of Staghorn, including its message passing system, virtual machine snapshotting, and network snapshotting. The snapshot was developed to be transparent to the distributed system being studied. In particular, any snapshot-specific artifacts that are introduced must be controlled for and as such should be limited. Additionally, we want the snapshotting to be efficient enough that it can be used repeatedly in order to accomplish state space exploration. This led to a few design requirements:

1. The system must not perceive that a snapshot has occurred
2. Staghorn must preserve machine state
3. Staghorn must preserve network state
4. Staghorn must snapshot quickly so that each virtual machine is snapshotted within a tight time window

Taking advantage of Sandia’s emulation-based modeling platforms, we built Staghorn on top of Firewheel [46] and Linux/QEMU/KVM/OVS. Staghorn consists of components that handle each aspect of taking a full-system snapshot. It contains a component to orchestrate the snapshot and ensure that it occurs within a tight time window. It contains a component to handle stopping virtual CPUs and saving the corresponding VM disk and memory. Finally, it handles capturing network packets in transit and providing mechanisms to modify, create new, and delete existing packets. The remainder of this chapter discusses the design decisions and implications for each of these components.

### 2.1 Full System Snapshot Staging

The staging system is designed to provide tight time windows across physical hosts to orchestrate and enact the snapshot.

#### 2.1.1 NTP vs PTP Time Synchronization

We were originally concerned about time synchronization of clocks across the cluster, as Staghorn depends heavily on them. We considered both NTP and PTP initially. NTP is a

globally available and easy to install system used on most Linux and Windows hosts. PTP is a specialized system that achieves sub-microsecond accuracy across connected systems, and relies on specially configured network interface cards. However, upon evaluating NTP we found that it can provide sub-millisecond accuracy across a LAN. The additional error introduced using NTP should be taken into consideration when applying Staghorn to latency-sensitive applications, however for development purposes as well as all Staghorn applications to date, NTP has been sufficient.

## 2.1.2 Fully-Distributed Snapshot Staging Architecture

The staging system is designed to provide a system level snapshot in as tight of a time window as possible. This is important as potentially thousands of virtual machines and logical networks will need to be snapshotted as close to simultaneously as possible. Accomplishing this hinges on the ability to execute commands at as precise a time as possible, in order to coordinate a cluster-wide snapshot. Our architecture accomplishes this by moving all time-sensitive components to execute in a very controlled manner. The criterion for determining when to snapshot is set in Staghorn as a trigger. This trigger could be a target time or the transmission of specific data over the network. An overview of the staging system architecture is shown in Figure 2.1. The user-facing sender component allows the definition of a trigger and the restoration of a snapshot.

The definition of a trigger is passed to one or more receivers using RabbitMQ. This provides a scalable, low-latency interface to pass data as messages between applications. Using a well-defined available protocol both simplifies the implementation and provides good scaling and robustness properties.

The receiver is responsible for receiving trigger definitions and taking action based on them. It remains less sensitive to execution time for these operations, and so is implemented in Python. The time sensitive component of the receiver, the Precise Staging Client, is implemented in C to minimize overheads. This component provides the actual snapshot implementation, and runs only after the receiver has a trigger definition.

For time triggers, the Precise Staging Client begins immediately, running a precise timer loop. This precise timer loop allows the snapshot to be triggered as close as possible to the desired time, without concerns about the precision of being woken from a long sleep call by the kernel.

The interface is slightly different for packet triggers. Some method of kernel to user space communication is needed, because the modified OVS kernel module performs the data matching for the trigger. Doing the matching in-kernel is the lowest latency option, avoiding the frequent copy of data to user space. This necessitates bidirectional communication with the kernel module. The use of a blocking syscall (so the user space process resumes when the syscall returns) was ruled out because implementation requires modifying the base kernel. This was undesirable because there was no other need to do this. Instead, we built a netlink

# Staghorn Staging System Overview

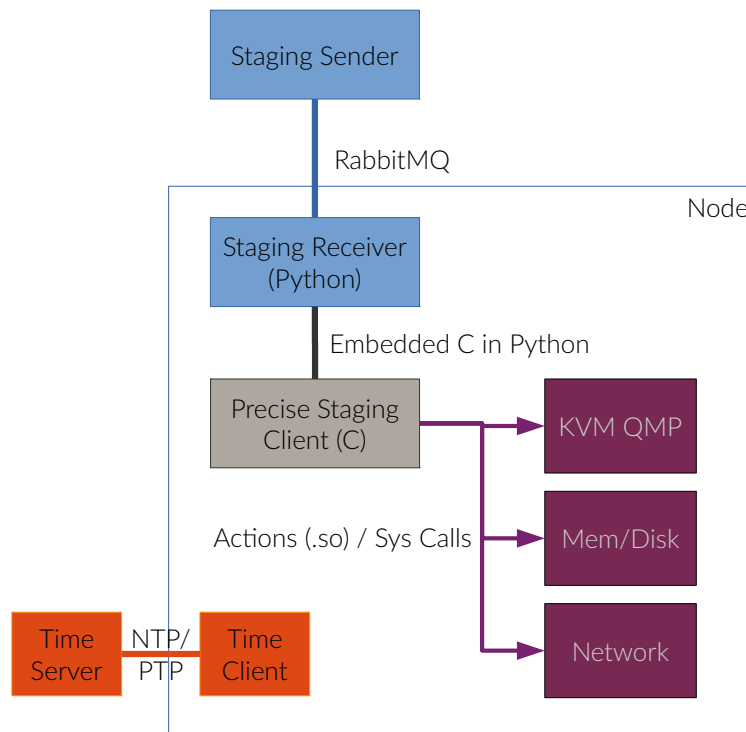


Figure 2.1: An overview of the Staghorn snapshot staging and messaging system.

socket mechanism for communication to define the trigger. Then, to return control to user space we use a signal. The receiver implements a handler for `SIGUSR1`, which triggers a message to all other receivers and an immediate snapshot.

The snapshot and restore operations are represented as a series of actions. Each of these actions is self-contained, and loaded by the precise staging client to prepare for a snapshot or restore operation. The interface to these actions allows long-running preparation code to run before it is time for the operation, allowing time sensitive operations to minimize what runs during the operation itself. For example, the action that stops VMs sets up its connections to QMP sockets during initialization, then simply sends the `stop` QMP command during the snapshot operation.

## 2.2 Virtual Machine Time Perception

One of the premises of Staghorn is that when the distributed system is snapshotted its behavior is correct independent of the snapshot and subsequent replays. An important part of that is that the snapshot is transparent to the virtual machine. One of the main artifacts of a snapshot to a virtual machine is a shift in time. As Staghorn is built on top of

QEMU/KVM, we needed to ensure that such a shift in time is undetectable in a VM.

Early testing of clock behavior in virtual machines showed some artifacts were present in time information when the VM underwent a migration snapshot. Our requirement for no perceived time passing required there to be no (or minimized) artifacts of this process.

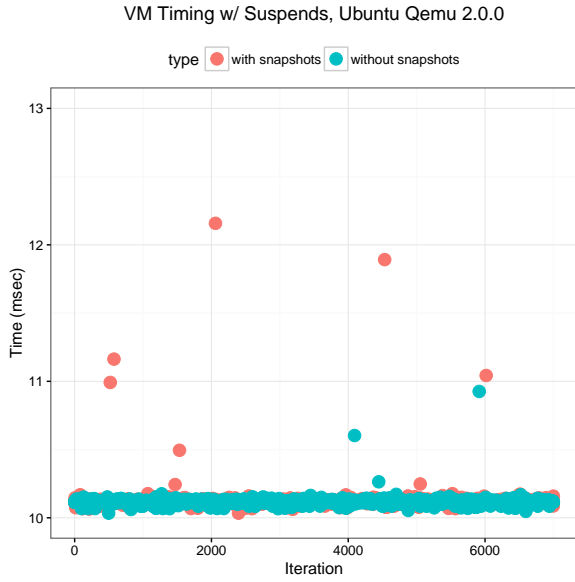
There are a lot of details when dealing with timekeeping in general [48], and when working with virtual machines there are even more complicating factors. Modern computers have a variety of time sources of varying precision, defined by the CPU architecture and standards like ACPI. The hardware abstraction required for virtual machines complicates the meaning of many traditional clock sources. The preferred clock source for Linux and Windows is the TSC, a counter defined by the Intel architecture to be monotonically increasing with each executed instruction. There are a variety of detailed considerations when using the TSC as a clock source, including the presence of different TSCs on different CPUs in multisocket systems. Our concern is the interaction of the virtualization systems with the TSC.

The TSC is accessed by the special `RDTSC` instruction. This causes a hardware VM exit on x86 systems with virtualization extensions [79]. For our purposes, this means QEMU contains the virtualized TSC code we are interested in. Notably, different hypervisors can handle the TSC differently—VMWare discusses their approaches in [18]. Further, TSC measurements are a common technique for detecting the presence of a VM, as discussed in [72].

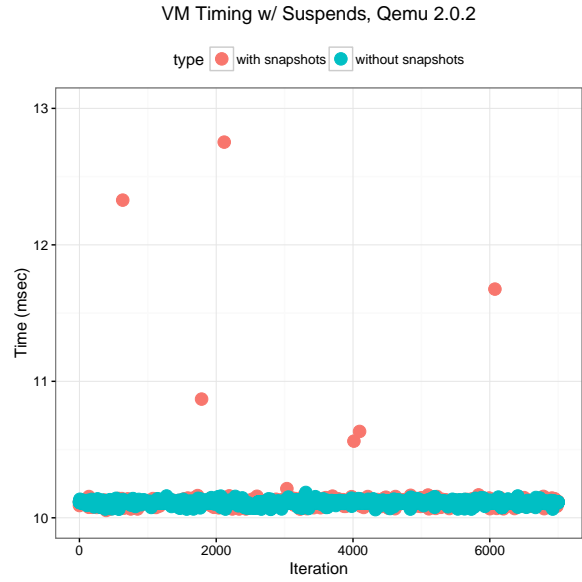
Our testing used the `gettimeofday()` system call, in a repeated loop after computing the square root of a random number. The goal is to run the test twice, once while snapshotting the VM (“with”) and once without performing any snapshots (“without”). Ideally, the output from “with” and “without” would be indistinguishable, meaning that the virtual machines cannot determine the snapshot has occurred. Working with QEMU 2.0.0, as included in Ubuntu 14.04 (before the patch [31]), we observed outlier cases where the time required for the loop increased 20-30% (2-3 ms) over the roughly 10 ms baseline time. While this measurement loop was running in the VM, a script would pause the VM (using the QEMU `stop` command) for 1 second every 5 seconds. The results of these tests are presented in Figure 2.2a.

The behavior was the same when using the upstream code for QEMU 2.0.2, as shown in Figure 2.2b.

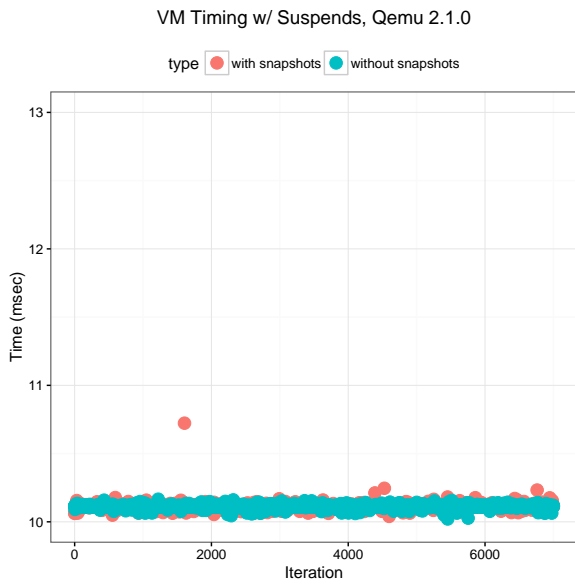
At the time of our work, there were recent changes to QEMU adjusting the behavior of the TSC during migrations [52, 28, 27]. Users were encountering TSC-related problems when trying to perform live migrations of virtual machines—the machine would hang, for example [31]. It is unclear exactly what change or changes lead to a fix for our observed behavior, but by QEMU 2.1.0 we were no longer observing the migration artifacts, although TSC changes continued with 2.1.3 [77]. The results of our microbenchmark on these QEMU versions can be seen in Figure 2.2c and Figure 2.2d. In these tests, there are no outliers beyond a fraction of a millisecond from the typical case. Consequently, upgrading QEMU solves the VM time perception problem.



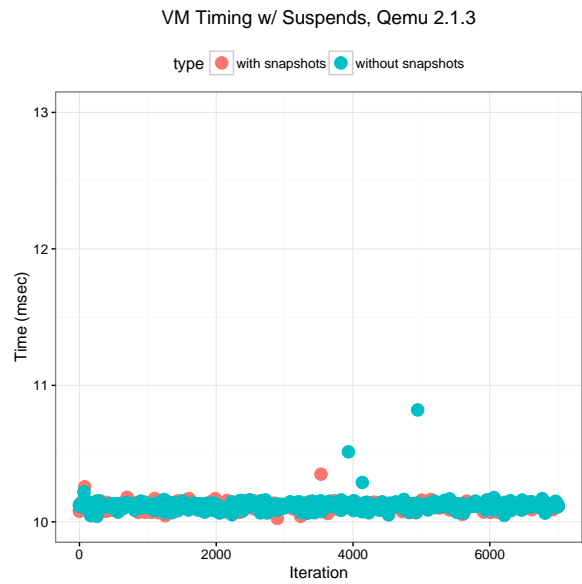
(a) Ubuntu QEMU 2.0.0. The snapshots are visible at approximately every 1000 iterations.



(b) QEMU 2.0.2. The snapshots are visible at approximately every 1000 iterations.



(c) QEMU 2.1.0. Even though snapshots are taken every 1000 iterations, they are not visible.



(d) QEMU 2.1.3. Even though snapshots are taken every 1000 iterations, they are not visible.

Figure 2.2: Timing tests measuring whether a virtual machine can detect a difference in its time due to being snapshotted. In these figures, “with” indicates that snapshots are being performed on the VM and “without” indicates that the VM is running without any snapshots being performed.

## 2.3 Virtual Machine State Snapshots

Staghorn requires taking snapshots of individual virtual machines that are running, including pausing them, saving their state, and returning and resuming from an arbitrary old state.

Three approaches were considered for virtual machine state snapshotting: QEMU migration-based snapshots, process-level snapshots, and QEMU fork-based snapshots. The stable implementation is based on QEMU migration-based snapshots, which are straightforward to implement because they utilize existing QEMU mechanisms. The two other mechanisms were explored and their implementation and design details are contained in Appendix B.

### 2.3.1 QEMU Migration Snapshots

QEMU implements a solution to preserve a VM at a given state—migration. This technique was developed to move a virtual machine between two physical hosts (hence the name “migration”), however it applies directly to saving a virtual machine (migrating to a file) and then restoring at a later point (migrating from a file). This technique has no code changes and is stable across QEMU upgrades. It does have higher overheads than some of the advanced techniques developed outlined in Appendix B, however these overheads do not outweigh the stability benefits. A VM migration snapshot is triggered with the QMP command `migrate` on a paused VM. This creates a file containing the hardware state of the VM: CPU, memory, etc. There are not many user-controlled options in this part of the process. We do have more flexibility, though, when considering how to preserve a consistent disk alongside the hardware state. We considered two options for this: QCOW2 disk files and LVM-based disks. Both options were implemented and both are available to end users.

#### QCOW2 Disks

QCOW2 is QEMU’s native virtual disk format. It represents a fully or sparsely provisioned disk as a single file. Snapshots may be created within the same file or as separate files (which reference a backing store). The snapshot files consist only of the data written since the snapshot was taken (they are copy-on-write). Such a snapshot can be applied to keep a consistent disk state with a migration snapshot.

#### LVM Disks

As an alternative to QCOW2 virtual disks, QEMU can use an LVM logical volume as a virtual disk. LVM includes a copy-on-write snapshot mechanism[93]. This is detailed in several resources, including [56, 14]. We set up a physical volume dedicated to hosting VM disks with 1 logical volume per VM. As a snapshot is taken, the virtual disk is preserved by



creating a new logical volume to store the changes from the previous. Execution can continue on the new logical volume with the previous logical volume becoming the permanent storage for a snapshot's disk state.

This use is a particular special case of LVM snapshots, where we pause the VM's execution. This allows us to bypass file system consistency concerns that otherwise come up when trying to perform operations on a running system [1].

Using LVM in place of QCOW2 requires preparing for VMs differently. The logical volumes must have data installed instead of simply copying a file. This is a straightforward step to add [30]. Additionally, LVM allows thin provisioning for logical volumes, so we do not lose any ability to run many VMs on a modest hard drive [68].

## 2.4 Snapshotting Networks

To accurately save and repeatably replay an entire distributed system's state, Staghorn is required to save network state including any in-flight packets. By saving in-flight packets, Staghorn is able to avoid challenges of missing data upon snapshot resume. The novel capability to save network state reduces the snapshots non-determinism and greatly improves the accuracy of replaying a snapshot. The following section outlines the design decisions our team made to enable capturing the full state of the network. The reader should note that the term packet is used a generic term for a layer-2 frame.

### 2.4.1 Design Considerations

#### Packet Latency and Ordering

An important consideration must be made when a snapshot is resumed. The packets that are captured during a snapshot necessarily were captured in a certain order and with a certain delay between them. When resuming, new packets may immediately be sent from a VM *and* packets may still need to be replayed from the captured buffer. This section considers the implications of two packet replaying techniques: one which preserves packet ordering for both the captured and new packets, we call packet ordering, and one which preserves the timing between each captured packet, we call packet spacing. We left the ultimate decision to the end user but default to preserving packet ordering.

Capturing and holding packets for an indefinite amount of time fundamentally breaks assumptions regarding packet ordering and latency of the distributed system under test. In particular, when in-flight packets are queued by Staghorn during a snapshot, it is possible that the additional queuing delay incurred by releasing the saved packets to OVS's data path is greater than the remaining components of the one-way delay outlined in Equation 2.1.

$$\begin{aligned}
OWD &= \text{One-way Delay,} \\
DT &= \text{Transmission Delay,} \\
DPP &= \text{Propagation Delay,} \\
DQ &= \text{Queuing Delay,} \\
DPC &= \text{Processing Delay}
\end{aligned}$$

$$OWD = DT + DPP + DQ + DPC \tag{2.1}$$

Figure 2.3 demonstrates the challenge of additional queuing delay. Suppose that the distributed system shown in Figure 2.3a is snapshotted such that the saved state is shown in Figure 2.3b. In this case packets  $P_B$ ,  $P_C$ ,  $P_D$ , and  $P_E$  have a queuing delay of  $QDS = QD_{initial} + QD_{staghorn}$  such that  $QDS_E > DT_F + DPP_F + DPC_F$ . This results in  $P_F$  arriving at OVS before  $P_E$  has been processed, as shown in Figure 2.3c. In this case, should the new packet endure the additional queuing delay placed upon OVS by Staghorn, or should the new packet move to the front of the queue? Additionally, should Staghorn preserve the spacing between packets as they arrive? Preserving the packet spacing could further increase the queuing delay and therefore, the possibility of the above problem.

Some possible unintended consequences of out-of-order packets include: triggering a TCP Negative Acknowledgment (NACK), reordering UDP packets potentially causing problems for an application, or sending an encrypted message when the protocol handshake has not completed yet. Additional latency can also have an impact on an experiment with latency-sensitive applications. By examining the various side effects, we determined that, by default, Staghorn should guarantee packet order and minimize queuing delay as much as possible. That is, the default behavior of Staghorn guarantees packet ordering but does not preserve packet spacing. However, Staghorn has the necessary code to preserve packet spacing and to promote latency over packet ordering should an experiment require these conditions. Appendix A explains our implementation that preserves packet spacing.

## Using Netlink

To pass information to and from our user code and the OVS kernel module, we had four options: system calls, `ioctl`s, proc file system, and netlink sockets. Our main design considerations for data passing were:

1. The system must be as quick as possible.
2. Staghorn must be easy to install.

Using system calls would require a modified kernel violating our “easy-to-install” requirement. Additionally, system calls are synchronous which can affect kernel scheduling and

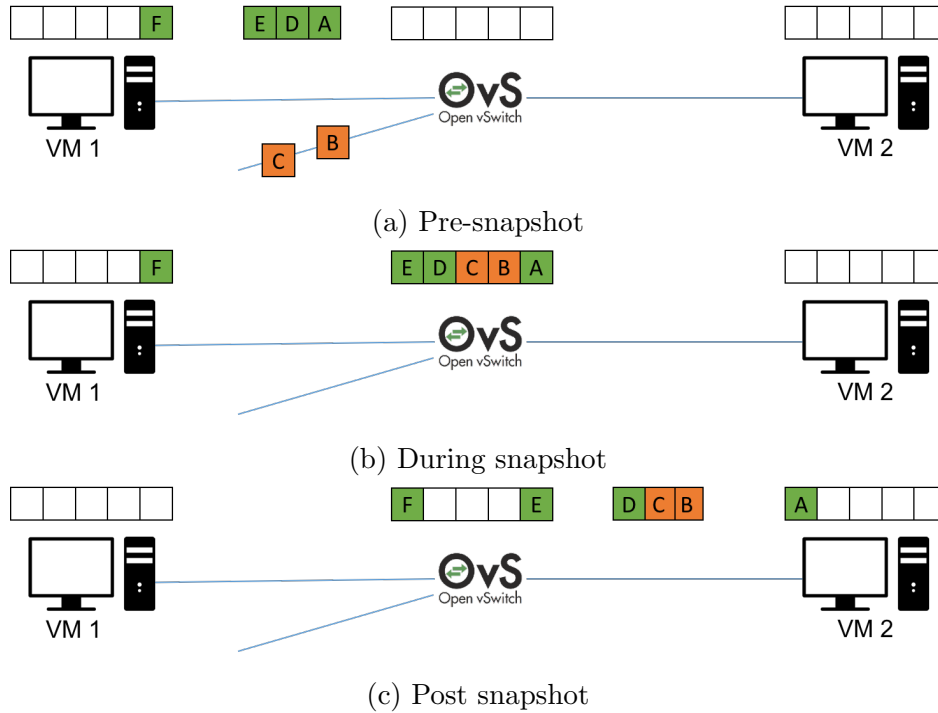


Figure 2.3: Example of the *OWD* introduced by Staghorn

increase latency when passing large data [57]. Both `ioctl`s and using `proc` as message passing are typically used via a polling interface. Polling adds latency to the process and places undue strain on the kernel [57]. In contrast, netlink is a quick asynchronous protocol that enables both the kernel and the user programs to initiate communication. Therefore, it was an obvious solution for passing data between Staghorn’s kernel and user level programs.

## Staghorn Modification Placement

We considered modifying the following system components to capture in-flight packets during a snapshot: separate hardware, Linux driver, and Linux Kernel/Open vSwitch. In looking at each of these components we had three main considerations.

1. All packets must be captured.
2. Staghorn must be easy to install.
3. It must work with an emulation platform.

Separate hardware or custom driver increases the difficulty of installing Staghorn on a commodity cluster. In addition, neither of these solutions would capture packets sent between two VMs on a single physical host. With these options eliminated we examined the Linux Kernel networking stack to find the best modification placement.

**The Socket Buffer** Each packet handled by the kernel is contained in a socket buffer structure (`struct sk_buff`), whose definition is found in `<linux/skbuff.h>` [38]. Though `sk_buff` structures, commonly abbreviated as `skb`, are used to store all network data throughout the Linux kernel, for the purposes of this paper we consider them only as individual packets [37, 23].

**Network Drivers** Receiving data from the network is trickier than transmitting it, because a `sk_buff` must be allocated and handed off to the upper layers from within an atomic context. There are two modes of packet reception that may be implemented by network drivers: interrupt driven and polled. However, most drivers implement the interrupt-driven technique [37].

Typically, an driver's interrupt receive function is called by the driver's interrupt handler once the packet has been processed by the physical device and is stored in memory. Receive function then potentially adds some additional information and passes it to the `netif_rx` function [37, 23]. The `netif_rx` function is responsible for handing the `skb` to upper layers of the networking stack. The `netif_rx` returns `NET_RX_SUCCESS` to indicate that the packet was successfully received and `NET_RX_DROP` means the packet was dropped. Packets could be dropped due to congestion control mechanisms or by some protocol layer [25].

**Open vSwitch** Open vSwitch (OVS) [11] is an open source virtual switch which was merged into the Linux kernel in version 3.3 [6, 35]. OVS is responsible for receiving a `skb` from `netif_rx` and then processing the packet. Figure 2.4 gives a general idea of how a packet progresses from `netif_rx` to OVS.

First, OVS needs to register a handler to inform the kernel to pass packets to it, rather than the regular networking stack. These handlers are defined in `datapath/linux/compat/netdevice.h` These handlers are defined in `datapath/linux/compat/netdevice.h` and the actual functions are located in `datapath/linux/compat/dev-openvswitch`. Each vport, or virtual port in OVS, device that OVS supports hooks the kernel differently. Some devices, such as, `vport-netdev` use the OVS hooks by calling the `netdev_create` function while others, such as `vport-internal`, call Linux kernel functions. Once the vports are hooked into the kernel, they can begin receiving packets. Currently OVS supports the following devices:

1. `vport-internal_dev.c`
2. `vport-netdev.c`
3. `vport-gre.c`
4. `vport-vxlan.c`
5. `vport-geneve.c`
6. `vport-lisp.c`
7. `vport-stt.c`

Most of the devices have a unique receive function in which they may perform some error

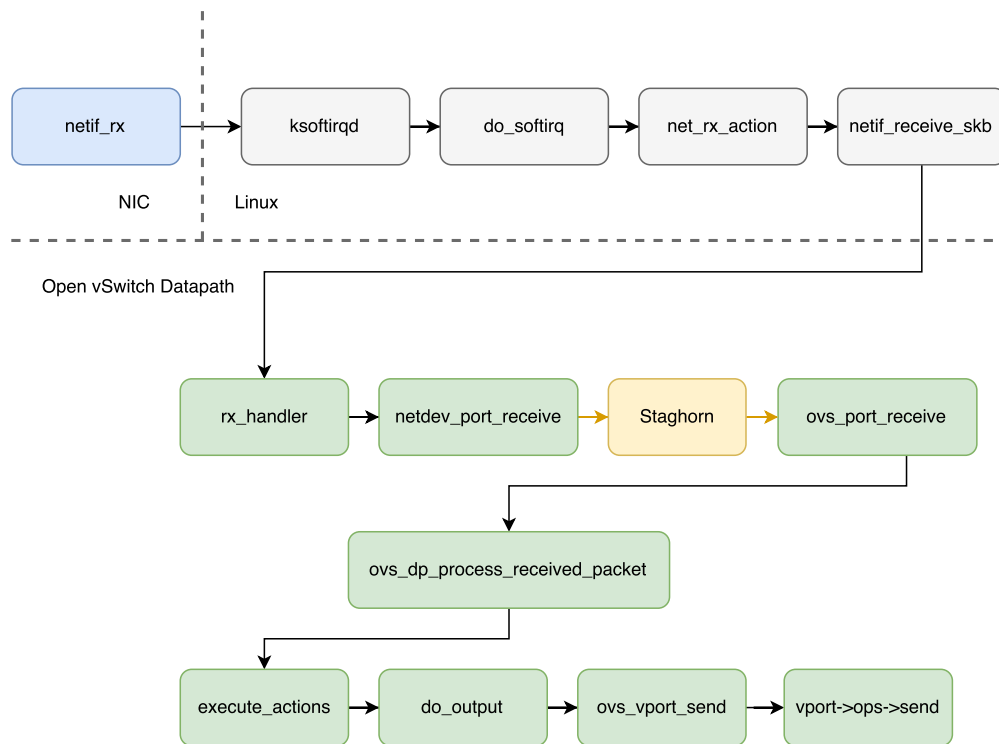


Figure 2.4: Rough outline of a packet’s progression to OVS [65]. The Staghorn entry and exit points are highlighted.

checking, such as verifying the checksum, or storing tunnel information. Internal devices (`vport-internal_dev`) are a special case because their receive function (`internal_dev_rcv`) passes the packet back to `netif_rx`. This is because internal devices are an OVS construct and cannot be accessed by the rest of the kernel [45, 98]. Therefore, when a packet is “sent” to an internal device, OVS receives it before the kernel’s `netif_rx` function. Once the packet returns to OVS, it is passed to `internal_dev_xmit`, which in turn “receives” the packet.

Each of OVS’s vports eventually call the `ovs_vport_receive` function, located in `datapathvport.c`, which passes the received packet to the datapath for processing. It must be called with `rcu_read_lock`. The packet cannot be shared and `skb->data` should point to the Ethernet header. The caller must have already called `compute_ip_summed()` to initialize the checksumming fields. This function sets packet statistics, initializes the inner data (converts it to network byte order), adds to the information in the OVS control block (`OVS_CB`<sup>1</sup>), extracts the flow key<sup>2</sup>, and sends the key and packet to the datapath to process (via `ovs_dp_process_packet`). Because `ovs_vport_receive` is last step before datapath and is called by all vports, it is an ideal predecessor to Staghorn modifications. When Staghorn is capturing packets it can return from the receive function and on resume it can pass the saved `skb` directly to `ovs_vport_receive`.

<sup>1</sup>This holds the packet as well as tunneling information and input vport.

<sup>2</sup>To find out more about the key see [73].

In addition to having an ideal spot for code modifications, OVS fits all desired criteria. It is compatible with many virtualization platforms including Xen, KVM, and VirtualBox, we are able to intercept all packets that enter/exit each virtual machine, its ubiquity makes it easy to install on new systems, and it works with two different Sandia-developed emulation platforms Minimega [10] and Firewheel [46].

# Chapter 3

## Implementation

This chapter describes implementation details of the various Staghorn components.

### 3.1 Full System Snapshot Staging

The staging system is implemented in two halves: the user-facing sender and the back-end receiver. An instance of the receiver runs on each physical host running VMs, and it is responsible for loading and executing actions. The sender provides a command-line interface to allow a user to define snapshot triggers and restore particular snapshots. The two components communicate using RabbitMQ, a message passing system. This provides scalable, low latency infrastructure to distribute a message among an arbitrary number of receivers.

The actions loaded and executed by the receivers represent the operation sequence. They are divided into two categories: critical and non-critical. Critical actions are run while the VMs are executing and packets are flowing across the wire. The execution time of critical actions controls the tightness of the operation. Non-critical actions are run after the system has been paused. Execution time is less important since no VM execution can take place while the actions are running. For snapshot operations, critical actions are run before non-critical actions. For restore operations, the order is reversed.

Each action is implemented as a shared library, callable from C, conforming to the action interface. The action interface consists of three functions, described in Table 3.1. The `init` function is called to prepare the action, before the operation has started. This allows longer running preparation to occur in critical actions, making their `run` function as short as possible. The `run` function is called while the snapshot or restore is occurring. An operation

Function	Arguments	Description
<code>init</code>	snapshot ID	Prepare to perform the action
<code>run</code>		Perform the action at operation-time
<code>destroy</code>		Clean up after the operation

Table 3.1: The interface functions for staging actions.

consists of a series of `run` functions. Finally, the `destroy` function is called to clean up after an operation has completed.

The receiver loads actions from the file system, with directories corresponding to the action and whether the action is critical, meaning its runtime impacts the correctness of the snapshot, or non-critical, meaning its runtime impacts the usability but not correctness. Actions for the snapshot operation are loaded from `/usr/lib/staghorn/actions/snapshot`, with the `critical` and `noncritical` subdirectories. Actions for the restore operation are loaded from `/usr/lib/staghorn/actions/restore`, with the `critical` and `noncritical` subdirectories. Actions are ordered, with the order determined by directory listing. As an example, the action to save off the COW disk changes since the last snapshot is in `/usr/lib/staghorn/actions/snapshot/noncritical/02-libkvmssavedisk.so`

### 3.1.1 Staghorn Triggers

The receiver only executes a snapshot operation based on a predefined trigger, not explicit user input. There are two types of triggers: time and packet. When a trigger is defined, the actions are loaded and their `init` functions executed.

When using a time trigger, the receiver runs a precise timing loop between the completion of initialization and the target time. This loop consists of a long sleep, followed by a short sleep, followed by a busy wait, and is necessary because no sleep mechanisms provided by Linux offer a precise enough wake-up.

A packet trigger makes a call (via netlink) to the in-kernel network snapshot system, passing the definition of the packet data to trigger on. It initializes the snapshot actions and waits to be triggered by the kernel. The kernel will send the receiver `SIGUSR1` when the packet data has been matched. From there, the receiver will send a RabbitMQ message so other hosts will also trigger a snapshot before starting its own snapshot process. The measured added delay from a packet-triggered snapshot is 1.29 milliseconds.

The trigger system was designed to be extensible and adding additional trigger types is anticipated to occur with use.

## 3.2 Virtual Machine Snapshots

To implement virtual machine snapshots, we created the `actions` in Staghorn's staging system found in Table 3.2. These actions create QMP sockets early during their initialization phase perform the necessary QMP handshake. Then, during their run phase, they send the appropriate message to carry out the action.

Each action is programmed to find the virtual machine QMP sockets that it connects to by walking the `/scratch/instances` directory.



Actions	Position	Description
<code>kvmstop</code>	snapshot, critical	Executes <code>stop</code> to quiesce VM CPUs
<code>kvmmigrate</code>	snapshot, non-critical	Executes <code>migrate</code> to move KVM data and buffers to user space and then into a file
<code>kvmquit</code>	snapshot, non-critical	Kills the <code>qemu</code> process
<code>kvmsavedisk</code>	snapshot, non-critical	Saves the QCOW2 disk file specific to this snapshot
<code>kvmrestoredisk</code>	restore, non-critical	Recovers the saved snapshot-specific QCOW2 disk file
<code>kvmrestoreconfig</code>	restore, non-critical	Restores all of the devices back into the VM
<code>kvmrestore</code>	restore, non-critical	Restores the migrate file back into the VM with <code>migrate-incomming</code>
<code>kvmstart</code>	restore, non-critical	Begins the KVM process
<code>kvmcont</code>	restore, critical	Starts the KVM CPUs which finishes the restore

Table 3.2: The actions used to snapshot VM state using QEMU’s migration-based snapshots.

The actions of the snapshot will work through the QEMU migration process, outlined in Section 2.3.1 in Chapter 2. Namely they will quiesce the CPUs in critical time, and then execute a migration, copy the disks, and shutdown the KVM process in non-critical time. A restore works through similar actions in the reverse direction, with starting the CPUs being the only action in critical time.

When Firewheel starts a VM it creates a script, `vm.sh`, containing the QEMU command it ran to start the VM, and a file, `qmp-cmds.log`, containing the QMP commands it issued. When restoring from a snapshot, the virtual machines are created based on the `vm.sh` script’s contents and then the initial QMP commands used to create the VM are again played into the new VM. To facilitate the numerous QMP sockets opened by each action, we modified Firewheel to launch VMs with a sufficient number of free QMP sockets.

### 3.3 Network Snapshots

To implement network snapshots, we created the `actions` in Staghorn’s staging system found in Table 3.3. These actions use a library we built called `netsnap` (network snapshot library) that communicates the desired snapshot action back into OVS. The actions will wait for a given return code before cleanly exiting.

We modified all of the `vport` calls to `ovs_vport_receive` to instead call the Staghorn replacement, `staghorn_enqueue`. This function evaluates what state Staghorn is in and

Actions	Position	Description
<code>netset</code>	snapshot, critical	Begins holding all packets in OVS in a snapshot-specific buffer
<code>netunset</code>	snapshot, non-critical	Stops holding all packets
<code>netstage</code>	restore, non-critical	Prepares the <code>netsnap</code> system, which communicates with the kernel, to replay
<code>netreplay</code>	restore, critical	Performs the actual replay of packets

Table 3.3: The actions used to snapshot network state.

handles the packet appropriately. We modified `datapath.c` to include our netlink sending and receiving, which includes features to extract packets from the kernel, load new packets into the kernel, replay packets from a particular snapshot, and take snapshots.

### 3.4 `netsnap_user` program

The `netsnap_user` program is used to communicate directly with the network snapshot system without using the staging system. This is used for building fuzzers, or other programs that need to pull packets from the kernel and load new packets back in. The implementation is split into a user program and a library so the staging system can make the same calls. The user program includes some calls not exposed through the staging system, namely the packet manipulation functionality.

For the user program, the usage is:

```
./netsnap_user <command> <snapshot name> [optional args]
```

Table 3.4 contains the possible commands, any optional arguments and their explanations. The snapshot name is an integer used to identify the snapshot.

The library, `libnetsnap`, includes functions implementing each feature exposed by `net-snap_user`, and some functions to initialize a netlink connection. These functions send one or more commands to the custom OVS kernel module, checking for a response before returning with a status code. A second library implements lower level netlink functionality, providing structures to track netlink sockets and packet buffers. This library is built for use through the higher-level `libnetsnap`. For operations related to snapshot and restore, the staging system loads the `libnetsnap` library to provide the necessary communications implementation.

Command	Optional Arguments	Explanation
SET	—	Start capturing packets (i.e. set a snapshot)
UNSET	—	Stop capturing packets
STAGE	—	Load the given snapshot
REPLAY	—	Start replaying packets
PACKETS	—	Save a <code>.sp</code> file (see Section 3.5)
PUT	[packet file]-the <code>.sp</code> file to inject	Put a <code>.sp</code> file back into the snapshot
TRIGGER	[pid]-the PID of the process listening for the signal; [offset]-the data offset; [data]-data on which to trigger	Trigger a snapshot on a given packet

Table 3.4: The commands used in `netsnap_user`.

## 3.5 Staghorn Packet File Format

Staghorn packet files (extension `.sp`) are created when users issue the `PACKETS` command to the `netsnap_user` program. They can be used to generate pcap files or can be modified to perform vulnerability analysis. These files contain the raw `struct staghorn_serialized_pkt_array` data structure shown in Code 3.1.

**Code 3.1** Raw structure of the `.sp` file.

```
typedef struct staghorn_serialized_pkt_array {
    size_t n_pkts; // Number of packets
    unsigned long snap_name; // Snapshot name
    unsigned char data[]; // Array of staghorn_serialized_pkt
} staghorn_serialized_pkt_array;
```

The `data` field is an array of `staghorn_serialized_pkt` structures, shown in Code 3.2.

**Code 3.2** Staghorn serialized packets.

```
typedef struct staghorn_serialized_pkt {
    struct staghorn_pkt_user_safe original_pkt_no_sk_buff; //
    Struct containing info used by OVS
    char sk_buff_shallow_bytes[SK_BUFF_SIZE]; // Critical non-
    data parts of the skb
    size_t sk_buff_len; // Total len of skb
    size_t data_offset; // Offset of skb data
    size_t data_size; // Size of skb data
    unsigned char sk_buff_data[]; // Socket buffer data
} staghorn_serialized_pkt;
```

## 3.6 Installation

Staghorn provides a series of scripts to help with the installation process. To begin, it assumes the Staghorn files are placed in `/opt/staghorn`. Installation requires setting up a few distinct pieces:

1. Precise timer
2. Actions
3. OVS
4. QEMU

First, to set up the precise timer, change directory to `/opt/staghorn/staging/precisetimer` and run `make`.

Next, to set up actions, change to `/opt/staghorn/staging/actions`. Run `make` and `sudo make install`.

Setting up OVS begins in `/opt/staghorn/network-snapshot`. Run `./install.sh` followed by `./dev-compile`. Then, change into `kernel\_comms\_netsnap` and run `make` then `sudo make install`.

Next, QEMU is in `/opt/staghorn/qemu`. QEMU has build dependencies that are not installed by default. Add them using `sudo apt-get build-dep qemu`. From there, it is a standard build process: `./configure`, `make`, `sudo make install`. Ensure that QEMU is not provided by the initial system and the binaries and files in `/usr/local` will be used.

### 3.6.1 Integration with Firewheel

Staghorn was designed to expect the same VM directory structure already used by Firewheel. This allowed Staghorn to run with Firewheel experiments with minimal modifications to Firewheel. Primarily, Firewheel was modified to provide a large number of QMP sockets for each VM. These sockets are then used by actions. Because of the `init/wait/run` architecture of actions, all actions that will run issue QMP commands maintain their own QMP connection and these connections are all active at the same time. Having many sockets for QMP allows all of these concurrent connections without any intermediate layer between QEMU and the actions.

For both Staghorn and other work, Firewheel was updated to use Open vSwitch for networking. In the case of Staghorn, this uses the custom OVS module. In Firewheel, OVS is selectable in place of the older Linux bridges networking implementation.

# Chapter 4

## Evaluation

In this chapter we evaluate the performance and characteristics of Staghorn, in addition to walking through several use cases that highlight how Staghorn can be used.

### 4.1 Benchmarking Studies

The benchmarks in this section were run on an HP ProLiant SL230s Gen8 with 128 GiB RAM, 512GiB Samsung 840 Pro SSDs, and dual socket E5-2670 cores at 2.6 GHz. The cluster has 10 GiBps networking setup and an Arista 7150S switch.

#### 4.1.1 Precisetimer test

To evaluate `precisetimer` that we developed, we built a test harness that tried to sleep one second into the future 60 times. It measured how close the sleep was to the desired sleep time and reported the error in nanoseconds. The benchmark can be found in Code 4.1.

**Code 4.1** A simplified form of the benchmarking code.

```
clock_gettime(CLOCK_REALTIME, &goal);
goal.tv_sec = current.tv_sec + 1;
precisetimer_sleepuntil(&goal, &res);
error = (res.tv_sec - goal.tv_sec)*1000*1000*1000 + (res.tv_nsec -
    goal.tv_nsec);
```

Overall, the results ranged from 1 nanosecond error to 55 nanoseconds, with a mean of 28.05. The results can be seen in Figure 4.1. This indicates that `precisetimer` is sufficient and is unlikely to introduce much additional sleeping overhead.

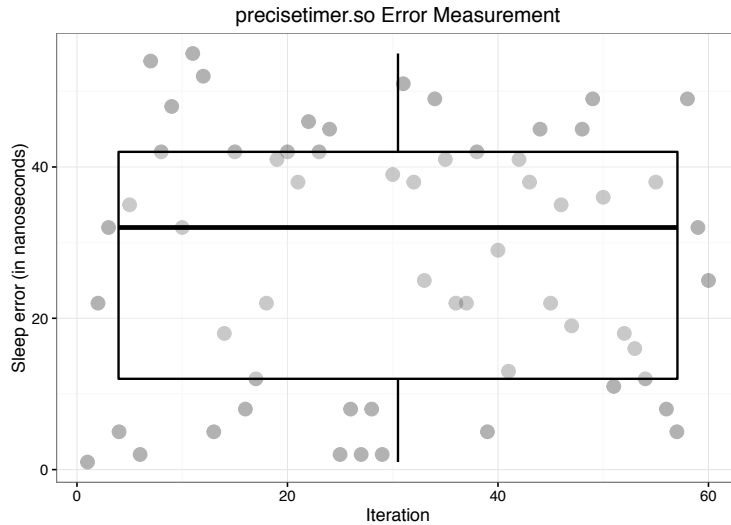


Figure 4.1: The results of the precisetimer test. Each iteration had a sleep target of one second into the future. The mean error was 28.05 nanoseconds.

### 4.1.2 RabbitMQ Message Speed

If a trigger is used that requires an immediate snapshot, the timeliness of RabbitMQ messages is important. To evaluate its performance, we measured the speed that RabbitMQ messages take on our cluster. To evaluate this we added a new message to Staghorn, `time`. This message results in all connected receivers running `pt.gettime()`. The sender additionally reports its `pt.gettime()`, and the difference between the two is measured. We performed this measurement on a local machine to control for any NTP-related clock drift issues and additionally on a remote machine. The mean was 1.29 milliseconds and the max was 1.87 milliseconds. The results can be seen in Figure 4.2.

This error is high enough that it should be noted when any trigger mechanism is used that relied on a RabbitMQ message to sent to cause the other physical machines to trigger. However, it is not so high as to break the correctness for many models, and could likely be improved with an alternate messaging system or a custom out-of-band trigger alerting mechanism.

### 4.1.3 Snapshot timing test

One of the most critical timing aspects of Staghorn is the performance of quiescing the virtual CPUs on each VM. To measure this, we isolated the actions to identify the time taken during the `kvmstop` action. The results of this evaluation can be found in Figure 4.3. The VMs are stopped in parallel through QMP. It is important to note that density and VM processing load will impact the ability for this to scale. Likely the workload will need to be

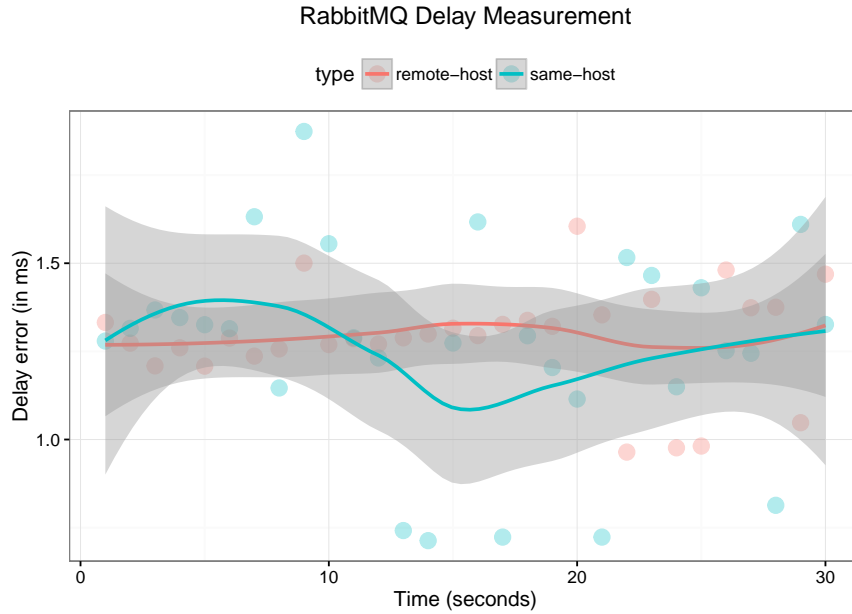


Figure 4.2: The results of the RabbitMQ introduced latency error test. The mean error was 1.26 milliseconds. The ribbon is a 95% confidence interval from the smoothing line. This test was performed both through RabbitMQ on localhost and on two machines within the testing cluster.

specified, and then the delay measured, to ensure that systems are not overscheduled and unable to snapshot rapidly enough. The mean snapshot time without loading contention was 2.555 ms.

#### 4.1.4 Dropped Packets

To ensure that Staghorn does not drop packets during a snapshot, we ran an experiment that flooded UDP traffic as quickly as possible between pairwise VMs. The scale of the experiment was varied from only two VMs total to over one hundred per physical host. Then, snapshots were made periodically during the traffic flooding. In all cases no packets were dropped.

#### 4.1.5 Guest VM Agnostic

We ran Staghorn on numerous underlying Firewheel models, varying between Windows clients, Linux clients, VyOS routers, emulated OVS switches, logical Firewheel switches, and handfuls of applications. In no case was the underlying model a factor in whether the snapshot was successful. However, with larger virtual machines, such as Windows 8.1 clients, the non-critical snapshot action that copies the hard drives took significantly longer.

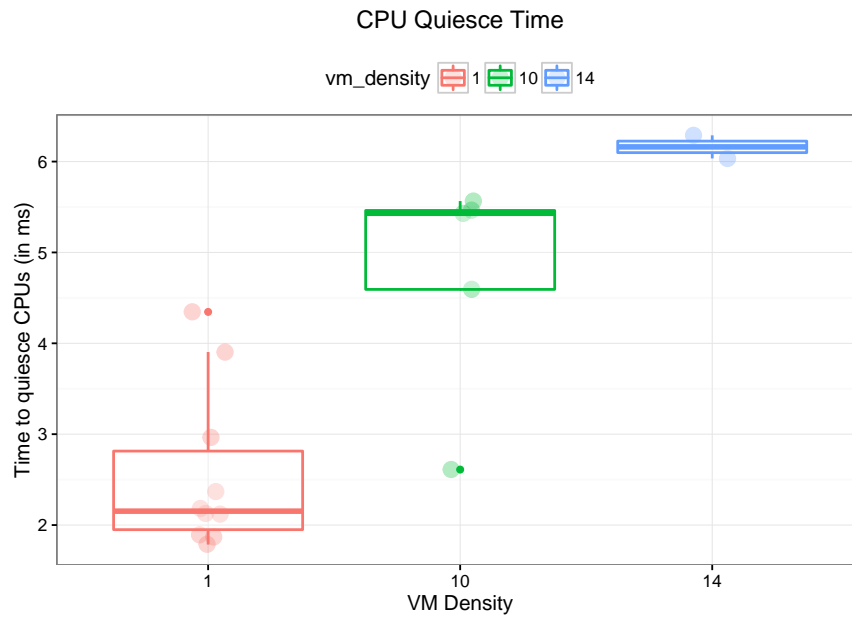


Figure 4.3: The results of measuring how long it takes quiesce a single CPU. As density and/or loading increases, each VM's CPUs will be quiesced in parallel but load and density is a limiting factor. This test was performed on a node with 16 cores, so any density higher than 15 (one core available to perform the snapshot) would not be recommended. This test contains measurement error as we are not capturing the moment when the CPUs are stopped but instead once we can query, out-of-band, that they are stopped.



## 4.2 Use Cases Tested

In this section, we evaluate a few of the use cases presented in Chapter 1.

### 4.2.1 Vulnerability Analysis

To evaluate Staghorn’s use in assisting with manual, semi-manual, or automated vulnerability analysis we chose to implement a distributed system fuzzer. This fuzzer will, using Firewheel and Staghorn, run an emulated model consisting of numerous virtual machines and execute it to an appropriate point to be fuzzed. Then, a network trigger is placed to create a snapshot at the moment the packet to be fuzzed is seen. Once the snapshot occurs, the system can be replayed numerous times while changing the packet that was captured. The system additionally is monitored and if a packet change results in an application crash then the packet is noted. An example of this framework in action can be found in Figure 4.4.

**Distributed Fuzzer** To demonstrate the added benefits of using Staghorn in conjunction with a network fuzzer, we developed a simple “dumb” fuzzer. Like other fuzzers, it is composed of three main components: a packet generator, a watcher, and a driver. The packet generator enables a user to create a set of fuzz cases that can be injected back into the experiment. It consumes a Staghorn packet file (`.sp`), described in Section 3.5 of Chapter 3, and modifies only the data specified. In our test implementation we found it useful to use Scapy [15] to perform packet parsing and modification. In our case, we generated 256 test cases in which a field was filled with random data. We also ensured that the generator would not modify any other `.sp` data. The watchdog component is a Firewheel agent that monitors the state of the application being fuzzed. The driver is responsible for testing each fuzz case against the desired snapshot. The driver iterates over each fuzz case and performs the following steps:

1. Insert packets into the snapshot
2. Resume the snapshot
3. Wait for the watchdog to respond
4. Note success/failure
5. Repeat with different test case

Once all of the test cases have run, the driver indicates if a particular case caused the controller to enter an error state or completely crash. If an error state is recorded, a user can replicate the same crash by sending the same fuzzed packet.

**Fuzzer Setup** To start fuzzing, we snapshotted to ensure that our system state is preserved precisely when it sees the first target packet. Therefore, we used a packet-based trigger

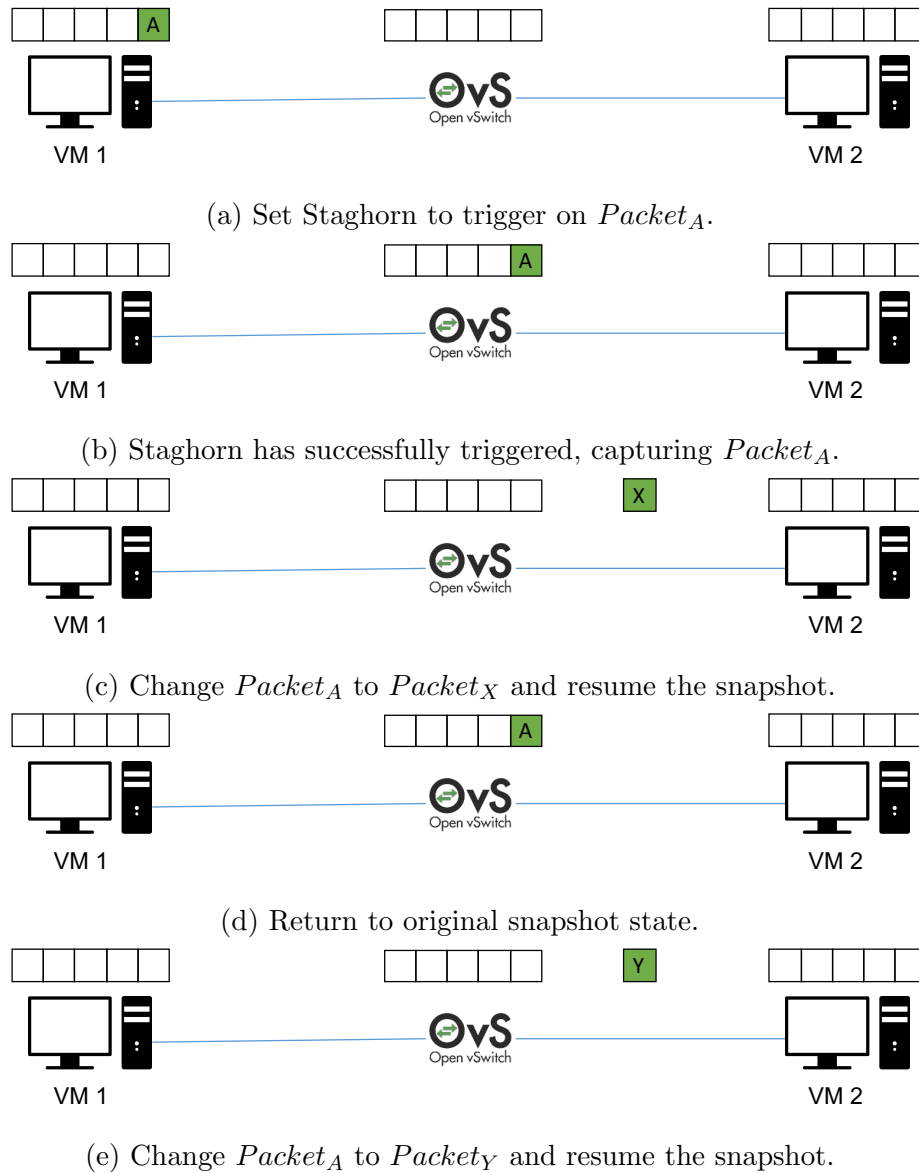


Figure 4.4: Staghorn can easily fuzz packets by triggering a snapshot on a target packet, modifying the packet, continuing the experiment, and returning to the pre-modification state

which, as detailed in Section 3.1.1 of Chapter 3 requires an offset and unique sequence of bytes located at that offset. For example, if the packet will contain 1234 at position 0x47, we could set the following trigger:

<b>Trigger</b>	
Offset	: 47
Data	: 1234

Once Staghorn has triggered on this packet we can invoke the fuzzer to generate our test cases.

Furthermore, the power of Staghorn can be used in conjunction with a “smart” fuzzer like Peach [13] to further extend the state-of-the-art in fuzzing.

## 4.2.2 Debugging Distributed Systems

### Remote Debugging

Using Staghorn, it is easy to stop an experiment when a error state occurs. This can be particularly useful when remotely debugging applications. While remote debuggers halt the application being debugged, this does not stop other entities from attempting to interact with the application. That is, there is no global breakpoint. Suppose an application is being debugged as part of a larger system, and it hits a breakpoint but none of the other applications communicating with it cease communication. This may impact the state of all of the other applications. Using Staghorn, a user can stop the entire experiment when a breakpoint is hit. Then the user can resume from the breakpoint as much as desired.

To demonstrate this capability we set ran an experiment with a Java client and a remote Java debugger. We attached the Java Debugger (`jdb`) and set a breakpoint in `Example.math`, shown in Figure 4.5. We then set the following trigger.

<b>Trigger</b>	
Offset	: 75
Data	: 4064020000000102

To determine the offset/bytestring we referenced the Java Debug Wire Protocol (JDWP) specification to find the correct offsets [4, 5]. The bytes map to:

1. `0x40` - Event Kind (Method entry)
2. `0x64` - Command set (Event command)
3. `0x02` - Suspend policy (Suspend all)
4. `0x00000001` - Number of events
5. `0x02` - Event kind (Breakpoint)

The breakpoint was triggered, as can be seen in Figure 4.6, and subsequently a snapshot occurred.

This is a trivial example to implement but also demonstrates the power of how Staghorn can augment remote debugging in which users are able to return to the exact state that caused the breakpoint to be triggered.

```
$ jdb -attach 198.128.0.1:8888
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> stop in Example.math
Set breakpoint Example.math
>
```

Figure 4.5: Setting a breakpoint in JDB.

```
Set breakpoint Example.math
>
Breakpoint hit: "thread=main", Example.math(), line=5 bci=0
main[1] █
```

Figure 4.6: The breakpoint being hit in JDB causing a snapshot to be triggered.

### 4.2.3 Debugging Emulated Experiments

To evaluate how helpful Staghorn is in debugging emulated experiments, we applied Staghorn to a problem in a Firewheel experiment. In this case the experiment had to run for around eight hours before a crash occurred. After the crash occurred, it was difficult to understand what happened and as tweaks were made, a wait time of eight hours had to be endured for every run. In this case Staghorn was shown to be able to snapshot the system shortly before the crash occurred, allowing a much quicker cycle for debugging.

# Chapter 5

## Conclusion

Distributed systems are becoming ubiquitous, increasing in complexity, and remain resistant to many previous analyses. Building on top of Sandia's large-scale emulation-based modeling platform Firewheel, we were able to develop the capability to take a full snapshot of a running model. This snapshot includes the network traffic in-transit and I/O of all virtual machines involved. This enables new classes of analyses to be run on distributed systems at scales that were not previously feasible. We developed the following components: an orchestration system which prepares the snapshot, techniques to capture full VM state, and a network snapshot and monitoring system which captures in-flight packets. Together, these components enable a user to preserve the entire state of an emulation experiment, explore it, modify it, and resume. Staghorn is already mature enough that it has been used in two efforts outside of its development project and is expected to be included in a future Firewheel release.

To demonstrate Staghorn's unique capability, we evaluated it for five applications. We performed a vulnerability analysis study that included fuzzing an application. We were able to return the entire system, including all of the virtual machines, networks, and software back to the pre-fuzz state for every packet. We remotely debugged a Java application and triggered a snapshot when it hit a breakpoint in its code. We evaluated its use in debugging experimental issues that occurred many hours into an experiment and identified how to use Staghorn to optimize regression testing and reduce or measure experimental non-determinism. Finally, we showed how Staghorn can improve a training environment by allowing participants to make mistakes without losing previous work.

Staghorn has opened the door to a host of new analyses. We anticipate continuing our optimization technique work to help improve state space exploration efficiency, improving our network packet modification system to provide more sophisticated control, and to integrate memory and disk introspection and modification tools.



# References

- [1] 3.8. snapshots. [Online]. Available: <http://tldp.org/HOWTO/LVM-HOWTO/snapshotintro.html>
- [2] “Google Code Archive - pagewalking.” [Online]. Available: <https://code.google.com/archive/p/pagewalking/>
- [3] How to use, monitor, and disable transparent hugepages in Red Hat Enterprise Linux 6? - Red Hat Customer Portal. [Online]. Available: <https://access.redhat.com/solutions/46111>
- [4] Java Debug Wire Protocol. [Online]. Available: <http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html>
- [5] Java Debug Wire Protocol Details. [Online]. Available: <http://docs.oracle.com/javase/6/docs/platform/jpda/jdwp/jdwp-protocol.html>
- [6] Linux 3.3 - Linux Kernel Newbies. [Online]. Available: [https://kernelnewbies.org/Linux\\_3.3/#head-d587af5a0e432c20cd96f2fe2b82adabba671df9](https://kernelnewbies.org/Linux_3.3/#head-d587af5a0e432c20cd96f2fe2b82adabba671df9)
- [7] Linux vm readme. [Online]. Available: <http://kos.enix.org/pub/linux-vmm.html>
- [8] Memory - KVM. [Online]. Available: <http://www.linux-kvm.org/page/Memory>
- [9] Microsoft Security Development Lifecycle. [Online]. Available: <https://www.microsoft.com/en-us/SDL/process/verification.aspx>
- [10] Minimega.org. [Online]. Available: <http://minimega.org/>
- [11] Open vSwitch. [Online]. Available: <http://openvswitch.org/>
- [12] Pagemap, from the userspace perspective. [Online]. Available: <https://www.kernel.org/doc/Documentation/vm/pagemap.txt>
- [13] Peach Fuzzer. [Online]. Available: <http://www.peachfuzzer.com/>
- [14] qemu-system-x86\_64(1). [Online]. Available: [http://man.cx/qemu-system-x86\\_64\(1\)](http://man.cx/qemu-system-x86_64(1))
- [15] Scapy. [Online]. Available: <http://www.secdev.org/projects/scapy/>
- [16] Split page table lock. [Online]. Available: [https://www.kernel.org/doc/Documentation/vm/split\\_page\\_table\\_lock](https://www.kernel.org/doc/Documentation/vm/split_page_table_lock)
- [17] The Definitive KVM (Kernel-based Virtual Machine) API Documentation. [Online]. Available: <https://kernel.org/doc/Documentation/virtual/kvm/api.txt>

- [18] Timekeeping in vmware virtual machines. [Online]. Available: <http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf>
- [19] “Transparent Hugepage Support.” [Online]. Available: <https://www.kernel.org/doc/Documentation/vm/transhuge.txt>
- [20] (29-May-2015) Information on the various kernel delay/sleep mechanisms. [Online]. Available: <https://www.kernel.org/doc/Documentation/timers/timers-howto.txt>
- [21] AF and Pm215. QOMConventions. [Online]. Available: <http://wiki.qemu.org/QOMConventions>
- [22] G. Banks, M. Cova, V. Felmetzger, K. Almeroth, R. Kemmerer, and G. Vigna, “SNOOZE: Toward a Stateful Network Protocol fuzZEr,” in *Proceedings of the 9th International Conference on Information Security*, ser. ISC’06. Springer-Verlag, pp. 343–358. [Online]. Available: [http://dx.doi.org/10.1007/11836810\\_25](http://dx.doi.org/10.1007/11836810_25)
- [23] C. Benvenuti, *Understanding Linux Network Internals*. O’Reilly.
- [24] R. Biro, F. van Kempen, M. Evans, and et al, “Linux/net/core/dev.c.” [Online]. Available: <http://lxr.free-electrons.com/source/net/core/dev.c>
- [25] R. Biro, F. van Kempen, M. Evans, F. La Roche, A. Cox, D. Hinds, A. Kuznetsov, A. Sulmicki, and P. Riikonen, “Linux/net/core/dev.c.” [Online]. Available: <http://lxr.free-electrons.com/source/net/core/dev.c>
- [26] E. Blake, “Re: [Qemu-devel] About VM fork in QEMU.” [Online]. Available: <https://lists.nongnu.org/archive/html/qemu-devel/2013-10/msg02912.html>
- [27] P. Bonzini, “[Qemu-devel] [PULL 5/6] Revert “kvmclock: Ensure proper env-`_tsc` value for `kvmclock_current_nsec` calculation”.” [Online]. Available: <http://lists.gnu.org/archive/html/qemu-devel/2014-07/msg02864.html>
- [28] —, “[Qemu-devel] [PULL 6/6] Revert “kvmclock: Ensure time in migration never goes backward”.” [Online]. Available: <http://lists.gnu.org/archive/html/qemu-devel/2014-07/msg02866.html>
- [29] —. QOM exegesis and apocalypse. [Online]. Available: <https://www.youtube.com/watch?v=fnLJn7PKhyo>
- [30] V. Botta. KVM LVM virtual machines: backups, cloning, and. [Online]. Available: <http://vitobotta.com/more-on-kvm-virtual-machine-host/>
- [31] P. Boven. guest hangs after live migration due to tsc jump. [Online]. Available: <https://bugs.launchpad.net/ubuntu/+source/qemu/+bug/1297218>
- [32] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O’Reilly.



- [33] M. Burdach. Physical memory forensics. [Online]. Available: <https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Burdach.pdf>
- [34] corbet. Improving page fault scalability. [Online]. Available: <http://lwn.net/Articles/114596/>
- [35] J. Corbet. Routing Open vSwitch into the mainline. [Online]. Available: <https://lwn.net/Articles/469775/>
- [36] —, “Transparent huge page reference counting.” [Online]. Available: <https://lwn.net/Articles/619738/>
- [37] J. Corbet, A. Rubini, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers*, 3rd ed. O’Reilly.
- [38] A. Cox and F. La Roche, “Linux/include/linux/skbuff.h.” [Online]. Available: <http://lxr.free-electrons.com/source/include/linux/skbuff.h>
- [39] A. Dabral. How is a page walk implemented? [Online]. Available: <https://www.quora.com/How-is-a-page-walk-implemented>
- [40] H. Dickins. How to use the Kernel Samepage Merging feature. [Online]. Available: <https://www.kernel.org/doc/Documentation/vm/ksm.txt>
- [41] U. Drepper. Memory part 3: Virtual Memory [LWN.net]. [Online]. Available: <http://lwn.net/Articles/253361/>
- [42] E. Dumazet, “[PATCH] NET : Convert Network Timestamps to Ktime\_t.” [Online]. Available: <https://www.mail-archive.com/netdev@vger.kernel.org/msg32710.html>
- [43] S. Eranian and D. Mosberger, “Virtual Memory in the IA-64 Linux Kernel.” [Online]. Available: <http://www.informit.com/articles/article.aspx?p=29961&seqNum=3>
- [44] A. Färber. Modern QEMU Devices. [Online]. Available: <http://www.linux-kvm.org/images/0/0b/Kvm-forum-2013-Modern-QEMU-devices.pdf>
- [45] T. Fifield, D. Fleming, A. Gentle, L. Hochstein, J. Proulx, E. Toews, and J. Topjian, *OpenStack Operations Guide*. O’Reilly.
- [46] K. Gabert, A. Vail, I. Burns, M. McDonald, S. Elliott, J. Montoya, J. Kallaher, T. Jones, and T. Thai, “Firewheel – A Platform for Cyber Analysis,” SAND2015-10324 PE.
- [47] T. Gleixner and I. Molnar, “Linux/include/linux/ktime.h.” [Online]. Available: <http://lxr.free-electrons.com/source/include/linux/ktime.h>
- [48] T. Gleixner and D. Niehaus, “Hrtimers and beyond: Transforming the linux time subsystems,” in *Proceedings of the Linux Symposium, Volume One*, pp. 333–346. [Online]. Available: <https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf>

- [49] S. Gorbunov and A. Rosenbloom, “Autofuzz: Automated network protocol fuzzing framework,” vol. 10, no. 8, p. 239.
- [50] M. Gorman, “Huge pages part 1 (Introduction).” [Online]. Available: <https://lwn.net/Articles/374424/>
- [51] —, “Understanding the Linux Virtual Memory Manager.” [Online]. Available: <https://www.kernel.org/doc/gorman/html/understand/>
- [52] A. Graf, “[Qemu-devel] [PATCH] kvmclock: Ensure time in migration never goes backward.” [Online]. Available: <http://lists.gnu.org/archive/html/qemu-devel/2014-05/msg00508.html>
- [53] M. E. Hoque, H. Lee, R. Potharaju, C. E. Killian, and C. Nita-Rotaru, “Adversarial testing of wireless routing implementations,” in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. ACM, 2013, pp. 143–148.
- [54] Intel. Intel(r) 64 and ia-32 architectures software developer’s manual. [Online]. Available: <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- [55] S. Jain, “Re: Difference between major page fault and minor page fault.” [Online]. Available: <https://www.mail-archive.com/kernelnewbies@nl.linux.org/msg09988.html>
- [56] Jared. KVM/qemu - use LVM volumes directly without image file? [Online]. Available: <http://serverfault.com/questions/410210/kvm-qemu-use-lvm-volumes-directly-without-image-file>
- [57] K. Kaichuan He. Kernel Korner - Why and How to Use Netlink Socket. [Online]. Available: <http://www.linuxjournal.com/article/7356>
- [58] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, “Finding protocol manipulation attacks,” *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 26–37, 2011.
- [59] R. Landley. Memory faq. [Online]. Available: <http://landley.net/writing/memory-faq.txt>
- [60] —. ramfs, rootfs, and initramfs. [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt>
- [61] M. Larabel, “Speeding Up The Linux Kernel With Transparent Hugepage Support - Phoronix.” [Online]. Available: [http://phoronix.com/scan.php?page=article&item=linux\\_transparent\\_hugepages&num=1](http://phoronix.com/scan.php?page=article&item=linux_transparent_hugepages&num=1)
- [62] H. Lee, J. Seibert, C. E. Killian, and C. Nita-Rotaru, “Gatling: Automatic Attack Discovery in Large-Scale Distributed Systems.” in *NDSS*, 2012.

- [63] T. Li, “[Qemu-devel] vm fork.” [Online]. Available: <https://lists.nongnu.org/archive/html/qemu-devel/2014-09/msg05072.html>
- [64] H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, “MODIST: transparent model checking of unmodified distributed systems.” NSDI, 2009.
- [65] lingmingyb, “OVS Datapath Processing.” [Online]. Available: <https://github.com/lingmingyb/ovs/wiki/Datapath-Processing>
- [66] MirkoBanchi, “Walking page tables of a process in Linux - Stack Overflow.” [Online]. Available: <http://stackoverflow.com/questions/8980193/walking-page-tables-of-a-process-in-linux>
- [67] I. Molnar, “Re: pte\_page.” [Online]. Available: <http://lkml.iu.edu/hypermail/linux/kernel/0105.3/1223.html>
- [68] mrwizer. LVM Thin Provisioning. [Online]. Available: <https://lxadm.wordpress.com/2012/10/17/lvm-thin-provisioning/>
- [69] S. Mugabi. Essential QEMU PCI API. [Online]. Available: [http://nairobi-embedded.org/001\\_qemu\\_pci\\_device\\_essentials.html](http://nairobi-embedded.org/001_qemu_pci_device_essentials.html)
- [70] M. Musuvathi, D. R. Engler *et al.*, “Model Checking Large Network Protocol Implementations.” in *NSDI*, vol. 4, 2004, pp. 12–12.
- [71] J. Novak and S. Sturges, “Target-Based TCP Stream Reassembly.” [Online]. Available: [http://webpages.cs.luc.edu/~pld/courses/intrusion/sum08/class5/novak\\_sturges.stream5\\_reassembly.pdf](http://webpages.cs.luc.edu/~pld/courses/intrusion/sum08/class5/novak_sturges.stream5_reassembly.pdf)
- [72] A. Ortega. rdtsc x86 instruction to detect virtual machines. [Online]. Available: <http://blog.badtrace.com/post/rdtsc-x86-instruction-to-detect-vms/>
- [73] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The Design and Implementation of Open vSwitch,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, pp. 117–130. [Online]. Available: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>
- [74] prosen. how to get struct vm\_area\_struct from struct page. [Online]. Available: <http://stackoverflow.com/questions/10265188/how-to-get-to-struct-vm-area-struct-from-struct-page>
- [75] S. Riku. Linux I/O port programming mini-HOWTO. [Online]. Available: <http://tldp.org/HOWTO/IO-Port-Programming.html>
- [76] C. Rohland. tmpfs. [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt>
- [77] M. Roth, “[Qemu-stable] [ANNOUNCE] QEMU 2.1.3 Stable released.” [Online]. Available: <http://lists.gnu.org/archive/html/qemu-stable/2015-01/msg00143.html>

- [78] D. A. Rusling. Chapter 3 - Memory Management. [Online]. Available: <http://www.tldp.org/LDP/tlk/mm/memory.html>
- [79] N. B. Sahgal and D. Rodgers, “Understanding Intel<sup>®</sup> Virtualization Technology (VT).”
- [80] M. Santosa. Getting confused when exploring Qemu source? gcc comes to rescue! [Online]. Available: <http://the-hydra.blogspot.com/2011/04/getting-confused-when-exploring-qemu.html>
- [81] R. Sasnauskas, P. Kaiser, R. L. Jukić, and K. Wehrle, “Integration testing of protocol implementations using symbolic distributed execution,” in *2012 20th IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2012, pp. 1–6.
- [82] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, “KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment,” in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 186–196.
- [83] C. Scott. Technologies for Testing Distributed Systems, Part I. [Online]. Available: <http://colin-scott.github.io/blog/2016/03/04/technologies-for-testing-and-debugging-distributed-systems/>
- [84] K. A. Shutemov, “[PATCHv5 00/28] THP refcounting redesign.” [Online]. Available: <http://lkml.org/lkml/2015/4/23/555>
- [85] —, “[PATCHv7 00/36] THP refcounting redesign.” [Online]. Available: <http://lkml.org/lkml/2015/6/23/303>
- [86] —, “[PATCHv8 05/32] rmap: support file thp.” [Online]. Available: <https://www.spinics.net/lists/kernel/msg2256018.html>
- [87] J. Song, T. Ma, C. Cadar, and P. Pietzuch, “Rule-based verification of network protocol implementations using symbolic execution,” in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011, pp. 1–8.
- [88] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi, “Can you fool me? towards automatically checking protocol gullibility,” in *In HotNets*. Citeseer, 2008.
- [89] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, ser. Artech House information security and privacy series. Artech House, oCLC: ocn213308372.
- [90] L. Torvalds, “Bug: Loops\_per\_jiffy based udelay() mostly shorter than requested.” [Online]. Available: <http://lists.infradead.org/pipermail/linux-arm-kernel/2011-January/038007.html>
- [91] D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Jcatascopia: monitoring elastically adaptive applications in the cloud,” in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 226–235.

- [92] R. Udiyar, “Transparent Hugepages in RHEL 6.” [Online]. Available: <http://www.slideshare.net/raghusiddarth/transparent-hugepages-in-rhel-6>
- [93] UnixArena. Linux LVM - How to take volume snapshot. [Online]. Available: [www.unixarena.com/2013/08/linux-lvm-how-to-take-volume-snapshot.html](http://www.unixarena.com/2013/08/linux-lvm-how-to-take-volume-snapshot.html)
- [94] user3337215, “C - usage of void put\_page(struct page \*page) in Linux - Stack Overflow.” [Online]. Available: <http://stackoverflow.com/questions/23848015/usage-of-void-put-pagestruct-page-page-in-linux>
- [95] R. van Riel, “mm/rmap.c.” [Online]. Available: <http://lxr.free-electrons.com/source/mm/rmap.c?v=3.18>
- [96] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huncyutt, “Vscope: middleware for troubleshooting time-sensitive data center applications,” in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 121–141.
- [97] C. Wang, S. P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi, “Performance troubleshooting in data centers: an annotated bibliography?” *ACM SIGOPS Operating Systems Review*, vol. 47, no. 3, pp. 50–62, 2013.
- [98] C. Wang. An Overview of Openvswitch Implementation. [Online]. Available: <http://wangcong.org/2012/10/20/an-overview-of-openvswitch-implementation/>
- [99] S. Weil, Alexander.stohr, Bradh, Vapier, Eblake, Pm215, P. Bonzini, Stefanha, Mkletzan, and Ajb. GettingStartedDevelopers. [Online]. Available: <http://wiki.gemu.org/Documentation/GettingStartedDevelopers>
- [100] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, “CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems.” in *NSDI*, vol. 9, 2009, pp. 229–244.



# Appendix A

## Packet Spacing

To conduct proper bookkeeping to ensure packet spacing during network snapshots, we needed to add a timestamp to the socket buffer. Upon snapshot resume, we take the difference of the packet timestamp and the time at which the snapshot occurs. This enables us to introduce the correct amount of latency before releasing the packet into the OVS datapath. Protocol specific timestamp options are not be affected by taking a snapshot as they are processed higher in the kernel. This appendix outlines how packets are timestamped within the Linux kernel, how and a discussion on kernel delay functions.

### A.1 Timestamps in the Linux Kernel

Network packet timestamps primarily involve either protocol options or the Socket Buffer data structure. This section will discuss both of these options and their impact on Staghorn.

#### A.1.1 SKB Time Fields

The socket buffer structure has a reserved field for a time stamp. The existing structure, as of Linux Kernel release 4.7, is shown in Code A.1 [38].

**Code A.1** Socket buffer timestamp fields [38].

```
struct sk_buff {
    union {
        struct {
            /* These two members must be first. */
            struct sk_buff      *next;
            struct sk_buff      *prev;
            union {
                ktime_t          tstamp;
                struct skb_mstamp skb_mstamp;
            };
        };
    };
}
```

```

    struct rb_node  rbnode; /* used in netem & tcp stack */
};
    ... // Rest of the skb struct
};

```

The `tstamp`<sup>1</sup> typically represents the time at which the packet was received [23]. In some cases it is used for the packet's scheduled transmission time [23]. Depending on the kernel version, `tstamp` field is set when `netif_rx` receives a packet and checks that the timestamp is set using the `net_timestamp_check` function [24]. This function then calls `__net_timestamp`, shown in Code A.3, located in `skbuff.h` [24]. The timestamps are use the `ktime_t` data structure which provides nanosecond resolution [47]. The `skb_mstamp` field is for multi resolution time stamps [38]. It's data structure is shown in Code A.2.

**Code A.2** Struct defining the `skb_mstamp` field [38].

```

struct skb_mstamp {
    union {
        u64      v64;
        struct {
            u32   stamp_us;
            u32   stamp_jiffies;
        };
    };
};

```

**Code A.3** Actual function to set the timestamp [38]

```

static inline void __net_timestamp(struct sk_buff *skb)
{
    skb->tstamp = ktime_get_real();
}

```

## A.1.2 Per-protocol timestamps

Many protocols have an optional or required timestamp field. Some of these include ICMP, IP, and TCP. The socket buffer timestamp is different than these protocol-specific stamps. The `skb tstamp` is only relevant for the system on which the `skb` is created, while the protocol stamps are transmitted as packet header data. Furthermore, these protocol stamps are set in the upper parts of the kernel (e.g. `net/ipv4/tcp_input.c`) and are set after the OVS `vport` handles the packet. Additionally, for some protocols, like TCP, the timestamp is not aligned with the system clock but rather a random value that can serve

<sup>1</sup>In older kernels, (roughly below 3.0) the timestamp field is described as `struct timeval stamp`. The effort to change it to `ktime_t` started around 2007 [42].



as an additional sequence number [71]. Therefore, it is highly unlikely, that any protocol specific timestamps will be impacted by taking a snapshot.

## A.2 Delays and Timing

To introduce accurate latency when resuming a snapshot, we must artificially delay execution before passing the stored packets to the OVS data path. To determine whether to use a sleep or a delay function, we must determine if the code is in atomic context [20]. In our case, both OVS and Staghorn uses spinlocks to protect globally accessible data. The use of spinlocks requires the kernel to not release the processor meaning atomic context should be used [37].

### A.2.1 Delaying

As mentioned above, because sleep functions release the process the `*delay` family of functions must be used in should be used in atomic context.

The delay functions are largely architecture-specific, though our setup has delay functions for nano, micro, and milli second delays. The main goal for each of these functions is to achieve at least the desired delay amount. However, that does mean that there could be more delay than requested [37]. Most of these delay functions use a software loop, each loop cycle taking delaying by the requested interval (nano, micro, or milli seconds). However, the actual implementations are extremely hardware dependent. The most common delay prototypes are shown in Code A.4.

**Code A.4** Common delay prototypes.

```
ndelay(unsigned long nsecs) // Nanosecond delay
udelay(unsigned long usecs) // Microsecond delay
mdelay(unsigned long msec) // Millisecond delay
```

In general, the `udelay` and `ndelay` have an upper bound on the requested delay amount [37]. This assists the user in choosing the delay function appropriate for their use case. For example, on our system the maximum argument for `ndelay` and `udelay` is 19,999. That is, `ndelay` cannot be used to delay for  $\geq 20\mu\text{s}$  and `udelay` cannot be used to delay for  $\geq 20\text{ms}$ . It is important to note that these delay functions are not designed for precision particularly due to the variability of CPUs. Interestingly, Linus Torvalds even noted that “historically, we’ve seen `udelay()` being **really** off (ie 50% off etc), I wouldn’t worry about things in the 1% range” [90]. He then goes on to suggest that if the code is sensitive to even a 5% error, the user picked a delay value that was too short [90]. Outside of the standard Linux delay functions, are other ways to accurately delay the kernel. The two most common methods include using port I/O and delaying with assembly instructions [75]. However, we decided that for packet-spacing purposes the `ndelay` and `udelay` functions would suffice.



# Appendix B

## VM State Snapshot Optimization

### B.1 Introduction

QEMU fork-based snapshots are an optimization of the state preservation process and remain a promising path with further investigation. Process-level snapshots were also pursued as a performance optimization, but were set aside in favor of fork-based snapshots due to the significant complexities encountered in the Linux virtual memory subsystem.

### B.2 Process-Level Snapshots

Process-level snapshots provide an in-kernel mechanism to preserve and restore the state of a particular process. In the case of QEMU, the process state includes the state of the virtual machine—QEMU manages and holds in its memory the virtual CPU state as well as the virtual machine’s main memory. Process-level snapshots use the copy-on-write (COW) semantics already used by the `fork()` syscall to provide a low-overhead copy of the memory representing a process state. Provided the process is not running on a CPU when the snapshot is taken, this represents the entirety of the process’ state.

Copy-on-write is a common and effective performance optimization. A traditional process snapshot takes place in a series of steps:

1. Stop process
2. Create a copy of each page in the process’ memory
3. Start process

Step 2 is resource intensive both in memory consumed and CPU time required to complete the operation. In contrast, copy-on-write reduces the resource usage of the snapshot process:

1. Stop process

2. Mark each page of the process' memory COW
3. Start process
4. When the process writes to a page, create a copy of that page and remap the process to use the new copy.

This procedure reduces the pages copied to the minimum that must be copied (because they differ) at any given time, and amortizes the time to copy those pages over the time interval they are all written to. Since a lot of a process' memory is code pages and other data that is static, this significantly reduces the amount of duplicate memory in use.

The implementation details of `fork()` prevent its direct use for these snapshots. Instead, snapshots are constructed beginning with the COW primitives. This approach requires a detailed understanding of the Linux virtual memory subsystem, which is covered next. Following that is a discussion of the design for process-level snapshots.

## B.2.1 Linux Processes

Linux structures virtual memory such that each process has its own virtual address space. This causes the process to be something of a basic unit when considering virtual memory. The Linux kernel does not directly handle processes and threads, instead dealing with its own construct of tasks. A task may represent a process, a thread, or a kernel thread<sup>1</sup> and is the basic construct scheduled on a CPU. Tasks are organized in a tree, rooted at the “init” process, and both user and kernel tasks are included in the tree. Each task may have an arbitrary number of children, and the structure of this tree is exposed as parent/child relationships in userspace.

For our purposes, we are interested in tasks representing processes. Different threads in the same process will share the same memory manager structure in their task data.

## B.2.2 Linux Virtual Memory Overview

Getting started with the Linux virtual memory subsystem, we found documentation to be spotty and often invalidated by (relatively) recent changes to the kernel. A large portion of this change has been driven by the shifts from 32-bit x86 to 64-bit x86\_64 and to multicore systems. Many code paths are the same, because there is backward compatibility in the architecture, but fundamental structures have changed to accommodate the new ability to support many processors and many gigabytes of memory (including changes like [34]). The best overview of many kernel systems available is still [32], which is now more than 10 years

---

<sup>1</sup>Kernel threads all share the same (kernel) address space, and so are discussed as threads and not processes.

old. Like many available sources, it works with the 2.6 kernel line and is losing relevance as the kernel continues to advance (in a few years many sections may be irrelevant). We found many answers in places like Stack Overflow were out of date. The best sources of documentation are the kernel's built-in documentation, but these are often narrowly focused on small subsystems with virtual memory [16, 19, 40]. We also found Gorman has written an e-book about more modern kernel systems [51]. Other recent resources include [41, 39, 78, 59]. There is also a notable presence of resources for IA64, which is similar to but distinct from our x86\_64 target [43].

Linux on x86\_64 uses paged virtual memory, and (as it did on x86) ignores segmentation. Many basic aspects of the virtual memory system are dictated by the architecture, but Linux provides its own abstraction layer that appears to target sharing as much code as possible between architectures. With the age of Linux and the Intel architecture, modern constructs can't escape the legacy of past mechanisms.

As defined by the architecture [54], the page table pointer is loaded into the cr3 register. There are 3 mechanisms to flush the TLB: write to the cr3 register, a dedicated instruction for a full flush, and an instruction to flush a single entry. Pages are normally 4KB in size, and the system also supports 2MB and 1GB pages. The architecture also provides a 4-level page table structure (changed from 2-level in x86) where each level of the page table is itself a 4KB page<sup>2</sup>. Processors support 48 bit VA or 38 bit PA (this also varies by model). Notably, each increase in page size removes 1 level from the depth of the page table for that page. Linux itself considers the middle two levels of page table optional, giving the code flexibility across architectures and allowing it to handle traditional x86, x86 with PAE (each address space is 32 bits, but can have more than 32 bits of physical memory), and x86\_64 with the same code. Correctly-written code uses access functions and macros to make these structural complexities transparent (the code is usually written for a 4-level table and collapses into no-ops whatever levels aren't present). When working with page tables, access involves physical address conversion and checks of flags encoded into the table entries in positions specified by the architecture (or occasionally Linux on system-defined positions.)

Linux divides memory into different classes, each with a distinct purpose. The two classes that are important for our work are the page cache and anonymous memory. Anonymous memory is data memory allocated by processes—the heap and stack data are anonymous memory. The name comes from the memory representing an unnamed address space—there is no named backing file for the data in memory. The page cache is used to cache file system I/O operations and special-purpose uses like tmpfs. Memory in different classes behaves differently, and have differing life cycles [86].

Linux implements an address space for each process. The kernel lives in the top-end of the virtual address space. On classic x86 systems, the kernel was at addresses above 0xC0000000, giving a 3GB user/1GB kernel virtual memory split. This is fixed at compile time.

---

<sup>2</sup>8 bytes per entry (pointer),  $4096 / 8 = 512$  entries per table,  $512 * 4096 = 2$  MB (the next larger page size)

When dealing with lower level kernel memory code, keeping virtual and physical addresses straight becomes important. Variables are often devoted to a specific type of address. Page tables (except the initial pointer) contain exclusively physical addresses. The kernel provides `__va()` and `__pa()` macros to assist in converting between virtual and physical addresses, given the CPU's current virtual address space, although other considerations apply to this conversion [67]. Kernel code or documentation also refers to virtual addresses as “linear addresses” in places. The phrase “physical page” is not commonly used, instead being called a “page frame”, where a “page” is always virtual.

As part of memory allocation and management, Linux maintains a list of all physical pages available in the system (so it naturally varies in size with the installed memory). On x86 systems, this was an array of pointers to struct pages. This array would always use about 1% of available memory and was always small enough to be walked linearly. These assumptions break down on x86\_64 systems—there are too many 4K pages when RAM sizes are tens of gigabytes or more. So, on 64-bit systems the global `mem_map` symbol still exists, but it is no longer a static, linear array.

### B.2.3 Memory Manager Data Structures

Starting at the `task_struct`, the top level of access to the memory management system is the `struct_mm`. This structure provides an overview of all allocated memory the process has access to. It also contains a `pgd_t` reference to the task's page table. The most interesting member of this structure is the `vm_area_struct` reference, which is a linked list of VM areas (which, notably, is neither circular nor doubly linked.) Each `vm_area_struct` represents a contiguous (virtually) allocated region of memory. A task may have any number of these up to 65,536, and they may be merged if they represent adjacent regions. Each `vm_area_struct` tracks a start and end virtual address, as well as flags and function pointers for operations that may be performed on the VM area. These function pointers include the page fault handler (`no_page`), changing permissions of member pages, etc. Also worth noting is that a `vm_area_struct` can represent a memory-mapped file [7]. Useful references in this area are [33, 51].

An alternative path leads to the `struct address_space`.

Memory management data structures are also used to compute the current memory usage of a task. This is complicated by having pages (e.g. shared library code) shared between tasks and other optimizations of the memory system. Task memory usage is discussed in terms of “virtual size” vs. the “real size” or RSS. The virtual size is the size of memory accessible through the task's `struct_mm` and could reasonably be thought of as validly accessible memory in the task's page table. The real size is the amount of memory allocated uniquely to this process. This is smaller than the virtual size.

The real size can be determined by walking the task's page table and counting valid pages. Separate interfaces to the memory manager exist to determine a task's virtual size.

Both are exposed to user mode, and the `ps` command will give both sizes. Some locations give one size or the other.

A helpful resource for examining page tables is the pagemap mechanism [12].

## B.2.4 Transparent Huge Pages

Linux exposes the multiple page sizes supported by `x86_64` (and presumably other architectures) through a feature it calls huge pages. Traditional huge pages are statically assigned at boot time and can be allocated from that static pool. They provide a useful performance improvement for particular, known workloads. On `x86_64`, huge pages may be 2MB or 1GB. Huge pages are not swappable like normal pages. Gorman provides a good introduction to huge pages in [50].

Since 2.6.38, Linux has included support for transparent huge pages, and they are enabled by default on both Ubuntu and RHEL [3]. These are dynamically created huge pages, managed by the `khugepaged` kernel thread. Transparent huge page (THP) support may be in one of 3 states: off, madvise, or on. When it is on, or when it is selective and a process triggers `khugepaged`, a transparent huge page may be created from a 2MB group of contiguous pages. These pages must all be on the same PTE table, because the merged THP will be accessed through a PMD entry that replaces the reference to the PTE level of the page table. Transparent huge pages are not used for 1GB pages. When THP is “on”, `khugepaged` periodically checks each page table for eligible regions to merge into huge pages. Transparent huge pages are swappable, and may be re-split when dealing with naive applications (although, as we found out, there are still ways to process page tables and break when THPs are present). As currently implemented, the THP merge mechanism is only applied to anonymous memory.

Transparent huge pages provide a performance improvement for workloads like we are likely to encounter with virtual machines [61, 92]. Part of this performance gain comes from reduced TLB misses and reduced translation lookups [92].

Ongoing kernel development has sought to reduce the number of conditions where a transparent huge page must be split, and to apply them more broadly to realize performance gains. As we were working, a significant revision of the transparent huge page feature was underway in the upstream kernel. These revisions reworked reference counting for THPs as well as per-process PMD splitting [84, 85]. These changes missed the merge window for 4.2, but were being actively developed and discussed with the community [36]. The ultimate goal of the developers was to move the transparent huge page system to be applicable to memory classes like the page cache, instead of just anonymous memory.

When dealing with transparent huge pages and different classes of memory, correctly managing reference counting becomes crucial. Linux has some conventions such as “`_get`” and “`_put`” around this that are important to understand [94]. Additionally, almost every operation has various levels of wrappers that include some degree of reference counting

assistance. Taking the time to understand the landscape is critical to getting working code.

### B.2.5 TMPFS

Tmpfs is a RAM-only filesystem provided by Linux. Tmpfs uses memory from the page cache as its backing store, and permits volumes with a fixed maximum size [76]. Creation is accomplished through the mount command.

We considered three options for RAM-based disks on Linux before deciding to use tmpfs: RAMfs, tmpfs, and RAMdisk. We concluded tmpfs was the most suitable for our workload, and we would not be giving up meaningful performance and would have mechanisms like a maximum filesystem size that are desirable [60].

The tmpfs system is implemented in page cache memory. At the time of our work, this meant that transparent huge pages did not affect it. We were not able to conclusively determine why this is, but it may be related to reference counting (because most of the changes to THP are focused on that) or low-level details of the page table used by tmpfs (e.g. PTE vs. PMD and the swap subsystem). A fuller understanding of the issues in this space would be helpful to continue work on in-kernel process snapshots.

### B.2.6 Hardware Virtualization

Hardware virtualization extensions include mechanisms for shadow page tables, which could affect our page table-based snapshot technique when using VMs. We performed some preliminary investigation into KVM [8] and these extensions [54]. Our implementation did not reach the point where issues of this nature would appear, so we remain unsure what effects would occur or how to resolve them.

### B.2.7 Snapshot Implementation

Our snapshot implementation took the form of added kernel code. We used a loadable module and introduced syscalls to interact with user space. Some debugging was accomplished by exposing information to `/proc`, but most information was communicated through the `dmesg` log. Along with our kernel code, we developed a series of automated test cases to exercise different scenarios as we encountered them. Our technique was developed sufficiently to handle basic processes and transparent huge pages, but not multi-threaded programs or QEMU.

In principle, our snapshot technique consisted of three operations: create a snapshot, restore a snapshot, and delete a snapshot. Each snapshot was composed of a full copy of the page tables for each task involved in the snapshot, and creating the snapshot marked



the memory COW for target processes. Restoring a snapshot replaced the tasks' page table pointer with the one saved previously. Done correctly, reference counting and COW would preserve data pages even in the event the process freed memory after the snapshot had been taken. Deleting a snapshot involved cleaning up any data pages that were only used by snapshots and cleaning up the snapshot accounting data structures. Notably, we preserve a reference to the page table, but do not preserve the `vm_area` structures, which contain virtual address ranges. This may cause problems if the process resizes its memory significantly, which we are not expecting for VM workloads (once the VMs are running).

Our development used a test process that allocated and write a repeating sequence (e.g. 'AAAAAA') to block of memory (at least one write is required to each page for correct test behavior [55]) before waiting for user input to continue. After input it would read back the contents of memory, and write a new sequence (the old value incremented by 1, e.g. 'BBBBBB'), repeating the process numerous times. Writing to the memory was important both to ensure that allocation and page table construction were fully complete, and to create an identifiable sequence to verify snapshot functionality. After a write sequence was complete, we could trigger a snapshot followed by another write sequence. After a successful restore, the next sequence read out would be the data from the first write, not the second. Our process used a 128 MiB block of data, and about 5 minutes after starting the process would have 129024 kB of transparent huge pages in use.

In addition to a human-interactive test process, we developed automated test scenarios along the same pattern. These scenarios triggered snapshot and restore operations themselves instead of relying on the user to inform the system that a snapshot had happened. The scenarios were set up to exercise different conditions, such as a linear progression of snapshots. Details about the different scenarios are provided in Table B.1.

The first developed component of the process-level snapshot was a full page table walk for a target process. We began this procedure manually, largely drawing on [66, 2] and the implementation of `/proc/<pid>/smaps`. We encountered a growing number of corner cases, and the code for this implementation expanded rapidly. This approach proved more useful, and our implementation became a series of callbacks that fit the existing walk. Intermediate stages of this walk that we depended on were removed in an updated version, with the reason given being that no code used the calls. It was straightforward to patch this code back into the newer kernel.

Our snapshot code executed in the environment of the calling process, which was a management process distinct from the snapshot target. In Linux, there are 2 processes widely known in the kernel: the current context (if it exists) and the init process. We located our target processes using a set of PIDs and walking the task tree from the init task. For each target process, we could get a page table reference from the memory management data structures associated with the task and start our page table walk.

Once we could find the entries associated with each page, we marked each as COW. Again, we began this process by manually setting flags in the page table entries before finding functions intended to fulfill this role. The functions addressed corner cases our code

Scenario	Description
Constant THP	Use madvise to enable THP for the process. Wait for THP to reach a steady state for the process, then follow the linear scenario.
Increasing THP	Use madvise to enable THP for the process. Take a single snapshot before transparent huge pages reach a steady state, and attempt to restore it.
Linear	Update memory state, take a snapshot, update memory state, restore snapshot. Evaluate current memory state to make sure snapshot actually restored before repeating the whole process (up to 50 times). Ignores the presence or absence of THP.
Repeat	Update memory state, take a snapshot, update memory state, restore snapshot. Evaluate current memory state to make sure snapshot actually restored. To repeat, simply update memory state before restoring the same snapshot again (up to 50 times). Ignores the presence or absence of THP.

Table B.1: The scenarios used in testing process-level snapshots

missed, and utilizing them improved our stability.

At this point, the cause of intermittent failures we were experiencing on simple processes was traced to transparent huge pages. With THP disabled across the system, our snapshot implementation was successfully handling our basic test process. However, this configuration was not desirable—our workload was expected to benefit from transparent huge pages. We were able to use the `/proc/meminfo` file to determine the THP usage of our target processes with the “AnonHuge” entry. By making our page table walk huge page aware and carefully handling PMDs as a final page table level where appropriate (they need to be marked as COW in place of PTEs, using distinct calls), we were able to successfully snapshot processes with THP. A corner case that we left unhandled was the restore of a process after transparent huge page usage had changed. We restored whatever state was present when we snapshotted, ignoring any THP creation or splitting that occurred in the interim time. In some cases, this appeared to leak transparent huge page allocations, although we did not track down the full implications of this approach. This leak occurred when we would restore a pre-THP-merge page table, and `khugepaged` would pass back through the process and create new transparent huge pages. If a new snapshot had not been taken that required the previously create THPs, the system would still believe they existed (as indicated by various accounting data), but there would be no way to reference them. We remain unsure if this affects only accounting data (e.g. `/proc/meminfo`) or if data structures controlling resource allocation are affected.

Threads introduce some complexity for our snapshot because they create a condition where more than one task shares a page table. We were not able to track down the errors we encountered when trying to apply our snapshots to threaded processes. Our snapshot only handles the minimum data structures necessary, and with threads we need to look into if we are tracking structures at too low of a level (e.g. track `vm_area` instead of page table) or not properly accounting for references to the same objects.

## B.2.8 Applying Technique to tmpfs

Even if we could get threaded processes working with our snapshots, we still needed to handle disk state. Migration-based snapshots would just create a copy-on-write addition to the disk, but being able to apply our copy-on-write memory snapshot could improve snapshot time. Additionally, questions remain about the VM’s I/O cache on the hypervisor, which presumably (like tmpfs) resides in the page cache.

Applying our snapshot to tmpfs memory required extensive modification. We removed checks for anonymous memory by altering our use of built-in functions to support our operations (being careful to preserve other checks that did not inhibit desired behaviors). We also had to locate the memory in use for a given tmpfs volume.

We first attempted to mark all memory on the system, marking them COW. This ends up crashing the kernel when applied to kernel memory (which includes the page cache).

Another approach to find tmpfs memory was similar to the approach to finding process

memory. We would walk mount points until we located the target tmpfs mount, and from there could reach the struct pages associated with the volume. This was sufficient to use rmap [95, 74] to locate page table entries. While this was expected to work, rmap ended up giving empty results.

The approach that ended up working involved walking from the task struct through its open file handles. This yields a radix tree of pages, which can be mapped back to page tables. These can then be marked COW like the anonymous memory we are accustomed to. Despite this marking, however, we were not able to observe any change when a snapshot was restored—something about COW was not working.

We believe this is related to the page fault handler for page cache memory. Since this memory is typically discarded instead of swapped, it would make sense that it also doesn't implement copy-on-write. We were not able to track down the page fault handler for the page cache to confirm this theory. It is also worth noting that regardless of configuration for anonymous memory, the page cache was found to always be utilizing huge pages. Another factor we did not track down was if the meaning of the OS-specific bits in page table entries changes based on the memory class. One concern in this area is we had to become more careful about checking and restoring individual PTEs based on the result of a `pte_present` call. This did not cause any (visible) problems with anonymous memory, but would cause crashes in tmpfs memory.

We found the page cache to contain subtle variations compared to working with anonymous memory. Among these was the function used to increment a page's reference count: it is `page_cache_get` in the page cache and `page_get` in anonymous memory. The latter performs additional checks before incrementing the counter.

Complicating tmpfs snapshots, some of the crashes after snapshot restore operations would be delayed. Often, the crashes would be related to reference counting problems. This represents something of a theme in our implementation—there are a lot of reference counters, and not a lot of documentation about how to do what where.

## B.3 QEMU Fork-based Snapshots

### B.3.1 QEMU Background

QEMU as an application fulfills two distinct rolls: VM management for accelerated virtual machines, and real-time instruction set translation and emulation. In the first roll, QEMU serves as a management interface for kernel services to facilitate the creation of hardware accelerated virtual machines (Xen or KVM). In the second, it allows users to run executables for different architectures on a single system. For snapshots, we are only concerned about hardware accelerated VMs. This limits the sections of QEMU source code that are meaningful. Resources for getting started with QEMU include [99]

QEMU uses an object model, the QEMU Object Model (QOM), to represent hardware. This replaces the older device structure, known as QDev. The transition is incremental, with some devices converted, some in progress, and others remaining as legacy devices. Additionally, QEMU represents basic computer classes like ACPI Intel PCs as machine classes, allowing it to support various types of embedded and single-board computers. Static analysis on QEMU source is complicated by a heavy dependence on macros to implement a custom object system in C. There are some sources of information on QOM [29, 21, 44, 69] and some resources for static analysis [80].

Over a series of releases, QEMU has built a migration mechanism from basic hardware state preservation into live migration between physical hosts with minimal VM downtime. This system is attractive from a snapshotting perspective because it already encapsulates the VM's state. Unfortunately, this system does not reconstruct internal data structure state in a way that is helpful for understanding requirements after a fork operation.

While QEMU does not need KVM to run, it is typically used as the user space component of a KVM VM. In this role, QEMU sets up and manages the hardware state for the VM, including memory and CPU state. KVM is configured by QEMU with settings and pointers for VM resources. Then, when the VM runs, QEMU callbacks are triggered for VM exits, including instructions like `rdtsc`. The code used by QEMU to integrate with KVM is not centralized, but instead distributed among the different components (such as virtual hardware implementations) where KVM integration is possible.

Interaction with KVM takes place starting through `/dev/kvm`, using IOCTL calls. As more KVM systems are configured, the process (QEMU in our case) will be given additional file descriptors to issue different classes of IOCTLs. The KVM API defines three such classes: system (issued to `/dev/kvm`), VM, and vcpu. Detailed documentation for the KVM API is provided with the Linux kernel [17].

### B.3.2 QEMU Snapshots

After exploring the possibility of Linux kernel modules for snapshots, we began looking at modifying QEMU to obtain the functionality we needed. The goal of writing a specialized snapshot process was to enable rapid snapshotting by avoiding the costly process of deep copying Virtual Machine (VM) memory. The fork system call provides a large piece of the functionality that we needed to create a fast snapshot process and the system call can be used from within QEMU. The concept of using fork in QEMU has been discussed previously by QEMU developers; however, developers decided that forking a VM has no production uses [26, 63]. Since there were no open source branches of QEMU that support fork, we decided to work towards creating a snapshot implementation that uses fork.

To determine the problems with using the fork system call in QEMU, we added a QEMU Machine Protocol (QMP) command that simply called fork. The original running VM continued to run with some side effects, such as the inability to display the desktop without

crashing. However, continuing execution of the child process crashed immediately. It soon became apparent that the KVM kernel module was maintaining state based in file private data for the process' handle to `/dev/kvm`. The state was configured through a series of ioctls at start up. Based on this information, our path forward to enabling fork was to recreate the kernel module's state by replaying the sequence of ioctls used to create the VM originally. We outlined the following tasks:

1. Trace all KVM ioctl calls
2. Map the KVM ioctl calls to the ioctl name
3. Develop the ability to read and replay a sequence of ioctl calls
4. Replay ioctls without modifying or replacing the memory preserved by fork
5. Continue one of the forked VMs

## B.4 KVM IOCTLS Traces

For debugging purposes, QEMU provides the ability to trace events. QEMU must be compiled with special configuration options to enable this functionality. The configuration to enable tracing is `--enable-trace-backends=simple`. To obtain a trace from QEMU, specify the events to be traced in a text file. The events we traced are:

- `kvm_vm_ioctl`
- `kvm_ioctl`
- `kvm_vcpu_ioctl`
- `kvm_device_ioctl`
- `qemu_anon_ram_alloc`
- `qemu_anon_ram_free`
- `qemu_vfree`
- `runstate_set`

This set of events allowed us to trace VM state changes, allocations, deallocations, and ioctls. Once QEMU was compiled to support event tracing, we ran QEMU with the following additional flag:

```
--trace events=events,file=events.trace
```

This produced a trace, but did not provide the names of the ioctls being called. To determine the ioctl names, we added an additional QMP command to print out all possible ioctl names and the hexadecimal value. Using this information, we produced a trace decoder that accepted a QEMU trace and the ioctl name mapping. It produced a chronological list of ioctls called. We also wrote a script to determine the unique ioctls that must be handled by our snapshot command after calling fork. The list of the ioctls is given below:

- KVM\_CREATE\_VM
- KVM\_TPR\_ACCESS\_REPORTING
- KVM\_SET\_SIGNAL\_MASK
- KVM\_SET\_REGS
- KVM\_GET\_MSR\_INDEX\_LIST
- KVM\_IOEVENTFD
- KVM\_X86\_GET\_MCE\_CAP\_SUPPORTED
- KVM\_SET\_IDENTITY\_MAP\_ADDR
- KVM\_SET\_MP\_STATE
- KVM\_SET\_LAPIC
- KVM\_CREATE\_IRQCHIP
- KVM\_SET\_SREGS
- KVM\_SET\_CPUID2
- KVM\_CREATE\_PIT2
- KVM\_SET\_XCRS
- KVM\_SET\_DEBUGREGS
- KVM\_IRQ\_LINE\_STATUS
- KVM\_CHECK\_EXTENSION
- KVM\_UNREGISTER\_COALESCED\_MMIO
- KVM\_SET\_USER\_MEMORY\_REGION
- KVM\_SET\_VCPU\_EVENTS
- KVM\_CREATE\_VCPU

- KVM\_REGISTER\_COALESCED\_MMIO
- KVM\_SET\_XSAVE
- KVM\_SET\_VAPIC\_ADDR
- KVM\_SET\_IRQCHIP
- KVM\_X86\_SETUP\_MCE
- KVM\_SET\_GSI\_ROUTING
- KVM\_SET\_MSRS
- KVM\_GET\_SUPPORTED\_CPUID
- KVM\_GET\_API\_VERSION
- KVM\_SET\_TSS\_ADDR
- KVM\_GET\_PIT2
- KVM\_GET\_VCPU\_MMAP\_SIZE
- KVM\_SET\_PIT2

## B.5 Replaying KVM IOCTLS

Using the information obtained about the list of required ioctls, we began trying to replicate these ioctl calls. This process was a combination of calling ioctls manually and calling existing QEMU functions that configure some part of the VM.

Our first step was to recreate the connection the KVM kernel module. QEMU provided a useful function called `kvm_init` that initialized that connection for us. After that point, we slowly started using the trace to replicate the VM state. We successfully added support for `KVM_SET_IDENTITY_MAP_ADDR` based on the QEMU code that originally called this function.



## DISTRIBUTION:

- 1 MS 0359 D. Chavez, LDRD Office, 1911
- 1 MS 0899 Technical Library, 9536 (electronic copy)





