USENIX Association

Proceedings of the 29th Large Installation System Administration Conference (LISA15)

November 8–13, 2015
Washington, D.C.

Conference Organizers

Program Co-Chairs
Cory Lueninghoener, *Los Alamos National Laboratory*
Amy Rich, *Mozilla Corporation*

Academic Co-Chairs
Paul Anderson, *University of Edinburgh*
Marc Chiarini, *MarkLogic Corporation*

Invited Talks Co-Chairs
Doug Hughes, *D. E. Shaw Research, LLC.*
Mario Obejas, *Retirado*

Lightning Talks Coordinator
Lee Damon, *University of Washington*

Workshops Chair
Lee Damon, *University of Washington*

Content Recruitment Team
Jonathon Anderson, *University of Colorado, Boulder*
Jennifer Ash-Poole, *NASA*
Patrick T. Cable, *MIT Lincoln Laboratory*
Travis Campbell, *Indeed*
Selena Deckleman, *Mozilla Corporation*
Matt Disney, *Oak Ridge National Laboratory*
Nicole Forsgren, *Chef*
Ski Kacoroski, *LOPSA*
Paul Krizak, *Qualcomm*
Tom Limoncelli, *Stack Exchange, Inc.*
Mike Rembetsy, *Etsy*
Chris St. Pierre, *Cisco Systems, Inc.*
Mandi Walls, *Chef*

USENIX Board Liaisons
David N. Blank-Edelman, *Apcera*
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*

USENIX Tutorials Staff
Natalie DeJarlais, *USENIX Association*
Rik Farrow, *USENIX Association*

LISA Lab Chairs
Tony Del Porto, *Cisco Systems, Inc.*
Andrew Mundy, *National Institute of Standards and Technology (NIST)*

LISA Build Coordinators
Branson Matheson, *sandSecurity*
Brett Thorson, *Cranial Thunder Solutions*

External Reviewers

Sara Alspaugh
Jonathon Anderson
Theo Benson
Charles Border
Patrick Cable
Travis Campbell
Stephan Fabel
Glenn Fink
Herry Herry
Doug Hughes
Brad Knowles
Adam Moskowitz
Steve Muir
Ari Rabkin
Yaoping Ruan
Josh Simon
Chad Verbowski
Hiroshi Wada
Tianyin Xu
Ding Yuan

LISA15: 29th Large Installation
System Administration Conference
November 8–13, 2015
Washington, D.C.

Message from the Program Co-Chairs v

Wednesday, November 11, 2015

Refereed Papers Session

Hyperprobe: Towards Virtual Machine Extrospection 1
Jidong Xiao, *College of William and Mary;* Lei Lu, *VMware Inc.;* Hai Huang, *IBM T. J. Watson Research Center;* Haining Wang, *University of Delaware*

Dynamic Provisioning of Storage Workloads 13
Jayanta Basak and Madhumita Bharde, *NetApp Inc.*

SF-TAP: Scalable and Flexible Traffic Analysis Platform Running on Commodity Hardware 25
Yuuki Takano, Ryosuke Miura, and Shingo Yasuda, *National Institute of Information and Communications Technology and Japan Advanced Institute of Science and Technology;* Kunio Akashi, *Japan Advanced Institute of Science and Technology;* Tomoya Inoue, *Japan Advanced Institute of Science and Technology and National Institute of Information and Communications Technology*

Spyglass: Demand-Provisioned Linux Containers for Private Network Access 37
Patrick T. Cable II and Nabil Schear, *MIT Lincoln Laboratory*

Poster Session

DevOps Is Improv: How Improv Made Me a Better Sysadmin 49
Brian Sebby, *Argonne National Laboratory*

Message from the LISA15 Program Co-Chairs

On behalf of the entire organizing committee, we welcome you to the 29th LISA conference and are excited to present this year's program and proceedings. LISA15 aims to help you architect and manage secure, scalable systems by offering a curated selection of 23 half- and full-day training sessions; 12 workshops; more than 60 talks, papers, presentations, and mini-tutorials; and an array of organized activities. We encourage you to engage with the field's top practitioners, share knowledge with your peers, and hone your skills. Don't forget to participate in the Birds-of-a Feather session and hallway conversations, too!

As LISA program chairs, one of our important charges is building and improving upon previous conferences. To continue providing a more streamlined and cohesive event, we followed last year's trend of coordinating the co-chairs for all of the sub-committees (training, workshops, talks, papers, and labs), not just the keynotes and papers. We also listened to your feedback and experimented with the addition of a new "content recruitment team." We approached a variety of well-connected industry leaders who could help identify authoritative presenters and suggest relevant material for the diverse and expanding field of computer operations.

We strove to provide an exceptional choice of subject matter and activities, covering numerous aspects of IT operations and engineering, to give you a unique, rewarding conference experience. As mentioned above, we rely on your feedback to let us know where we've done well and where you'd like to see change. Please be sure to fill out the training and/or conference attendee surveys in order to help us deliver a quality event you're eager to attend.

While the program chairs act as the face of the conference, it takes an amazing team of people to pull it all together. We are extremely proud of LISA15 and would like to thank everyone involved in making it a success: our subcommittee co-chairs, content recruitment team, paper reviewers and shepherds, LISA Build team, USENIX Board liaisons, USENIX staff, presenters, sponsors, vendors, and of course, you, our attendees.

We hope you have a fantastic conference!

Amy Rich, *Mozilla Corporation*
Cory Lueninghoener, *Los Alamos National Laboratory*
LISA15 Program Chairs

Hyperprobe: Towards Virtual Machine Extrospection

Jidong Xiao
College of William and Mary
jxiao@email.wm.edu

Lei Lu
VMware Inc.
llei@vmware.com

Hai Huang
IBM T.J. Watson Research Center
haih@us.ibm.com

Haining Wang
University of Delaware
hnw@udel.edu

Abstract

In a virtualized environment, it is not difficult to retrieve guest OS information from its hypervisor. However, it is very challenging to retrieve information in the reverse direction, i.e., retrieve the hypervisor information from within a guest OS, which remains an open problem and has not yet been comprehensively studied before. In this paper, we take the initiative and study this reverse information retrieval problem. In particular, we investigate how to determine the host OS kernel version from within a guest OS. We observe that modern commodity hypervisors introduce new features and bug fixes in almost every new release. Thus, by carefully analyzing the seven-year evolution of Linux KVM development (including 3485 patches), we can identify 19 features and 20 bugs in the hypervisor detectable from within a guest OS. Building on our detection of these features and bugs, we present a novel framework called Hyperprobe that for the first time enables users in a guest OS to automatically detect the underlying host OS kernel version in a few minutes. We implement a prototype of Hyperprobe and evaluate its effectiveness in five real world clouds, including Google Compute Engine (a.k.a. Google Cloud), HP Helion Public Cloud, ElasticHosts, Joyent Cloud, and CloudSigma, as well as in a controlled testbed environment, all yielding promising results.

1 Introduction

As virtualization technology becomes more prevalent, a variety of security methodologies have been developed at the hypervisor level, including intrusion and malware detection [26, 30], honeypots [48, 31], kernel rootkit defense [42, 40], and detection of covertly executing binaries [36]. These security services depend on the key factor that the hypervisor is isolated from its guest OSes. As the hypervisor runs at a more privileged level than its guest OSes, at this level, one can control physical resources, monitor their access, and be isolated from tampering against attackers from the guest OS. Monitoring of fine-grained information of the guest OSes from the underlying hypervisor is called virtual machine introspection (VMI) [26]. However, at the guest OS level retrieving information about the underlying hypervisor becomes very challenging, if not impossible. In this paper, we label the reverse information retrieval with the coined term virtual machine extrospection (VME). While VMI has been widely used for security purposes during the past decade, the reverse direction VME—the procedure that retrieves the hypervisor information from the guest OS level—is a new topic and has not been comprehensively studied before.

VME can be critically important for both malicious attackers and regular users. On one hand, from the attackers' perspective, when an attacker is in control of a virtual machine (VM), either as a legal resident or after a successful compromise of the victim's VM, the underlying hypervisor becomes its attacking target. This threat has been demonstrated in [35, 21], where an attacker is able to mount a privilege escalation attack from within a VMware virtual machine and a KVM-based virtual machine, respectively, and then gains some control of the host machine. Although these works demonstrate the possibility of such a threat, successful escape attacks from the guest to the host are rare. The primary reason is that most hypervisors are, by design, invisible to the VMs. Therefore, even if an attacker gains full control of a VM, a successful attempt to break out of the VM and break into the hypervisor requires an in-depth knowledge of the underlying hypervisor, e.g., type and version of the hypervisor. However, there is no straightforward way for attackers to obtain such knowledge.

On the other hand, benign cloud users may also need to know the underlying hypervisor information. It is commonly known that hardware and software systems both have various bugs and vulnerabilities, and different hardware/software may exhibit different vulnerabilities. Cloud customers, when making decisions on the choice of a cloud provider, may want to know more informa-

tion about the underlying hardware or software. This will help customers determine whether the underlying hardware/software can be trusted, and thus help them decide whether or not to use this cloud service. However, for security reasons, cloud providers usually do not release such sensitive information to the public or customers.

Whereas research efforts have been made to detect the existence of a hypervisor [25, 22, 24, 50], from a guest OS, to the best of our knowledge, there is no literature describing how to retrieve more detailed information about the hypervisor, e.g., the kernel version of the host OS, the distribution of the host OS (Fedora, SuSE, or Ubuntu?), the CPU type, the memory type, or any hardware information. In this paper, we make an attempt to investigate this problem. More specifically, as a first step towards VME, we study the problem of detecting/inferring the host OS kernel version from within a guest OS, and we expect our work will inspire more attention on mining the information of a hypervisor. The major research contributions of our work are summarized as follows:

- We are the first to study the problem of detecting/inferring the host OS kernel version from within a VM. Exploring the evolution of Linux KVM hypervisors, we analyze various features and bugs introduced in the KVM hypervisor; and then we explain how these features and bugs can be used to detect/infer the hypervisor kernel version.
- We design and implement a novel, practical, automatic, and extensible framework, called Hyperprobe, for conducting the reverse information retrieval. Hyperprobe can help users in a VM to automatically detect/infer the underlying host OS kernel version in less than five minutes with high accuracy.
- We perform our experiments in five real world clouds, including Google Compute Engine [3], HP Helion Public Cloud [29], ElasticHosts [20], Joyent Cloud [8], and CloudSigma [19], and our experimental results are very promising. To further validate the accuracy of Hyperprobe, we perform experiments in a controlled testbed environment. For 11 of the 35 kernel versions we studied, Hyperprobe can correctly infer the exact version number; for the rest, Hyperprobe can narrow it down to within 2 to 5 versions.

2 Background

Hypervisor, also named as virtual machine monitor, is a piece of software that creates and manages VMs. Traditionally, hypervisors such as VMware and Virtual PC use the technique of binary translation to implement virtualization. Recently, x86 processor vendors including Intel and AMD released their new architecture extensions to support virtualization. Those hypervisors that use binary translation are called software-only hypervisors, and recent hypervisors that take advantage of these processor extensions are called hardware assisted hypervisors [12]. In this paper, we focus on a popular hardware assisted commodity hypervisor, Linux KVM. We develop our framework and perform experiments on a physical machine with Linux OS as the host, which runs a KVM hypervisor, and a VM is running on top of the hypervisor. Our study covers Linux kernel versions from 2.6.20 to 3.14. While 2.6.20, released in February 2007, is the first kernel version that includes KVM, 3.14, released in March 2014, is the latest stable kernel at the time of this study. More specifically, we study the evolution of KVM over the past seven years and make three major observations. In this section, we briefly describe Linux KVM and report our observations.

2.1 Linux KVM

KVM refers to kernel-based virtual machine. Since Linux kernel version 2.6.20, KVM is merged into the Linux mainline kernel as a couple of kernel modules: an architecture independent module called kvm.ko, and an architecture dependent module called either kvm-intel.ko or kvm-amd.ko. As a hardware assisted virtualization technology, KVM relies heavily on the support of the underlying CPUs and requires different implementations for different CPU vendors, such as Intel VT-x and AMD SVM. Figure 1 illustrates the basic architecture of KVM. KVM works inside a host kernel and turns the host kernel into a hypervisor. On top of the hypervisor, there can be multiple VMs. Usually KVM requires a user-level tool called Qemu to emulate various devices, and they communicate using predefined ioctl commands.

Over the years, KVM has changed significantly. The original version in 2.6.20 consists of less than 20,000 lines of code (LOC); but in the latest 3.14 version, KVM modules consist of about 50,000 LOC. The reason of such growth is that 3485 KVM related patches have been released by Linux mainline kernel[1]. By carefully analyzing these patches, we make a few important observations about the evolution of the KVM development process.

First, while ideally hypervisors should be transparent to guest OSes, this is not realistic. In particular, during its development process, on the one hand, KVM exposes more and more processor features to a guest OS; on the other hand, KVM has been provided with many paravirtualization features. These changes improve performance but at the cost of less transparency.

Second, for the sake of better resource utilization, KVM has also included several virtualization-specific features, e.g., nested virtualization [16] and kernel same

[1] KVM has recently started supporting non-x86 platform, such as ARM and PPC; however, in this study, we only consider patches for x86 platforms, i.e., the number 3485 does not include the patches for the non-x86 platforms.

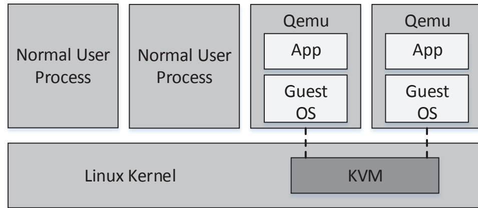


Figure 1: KVM Overview

page merging (KSM) [15], many of which can be detected from within the guest OS.

Third, similar to all other large projects, KVM have bugs. Among the 3485 patches, about 30% of them are bug fixes. In particular, we notice that a common type of bugs in KVM is related to registers. This reflects the fact that emulating a CPU is hard. Since a modern CPU defines hundreds of registers, emulating the behaviors of various registers correctly is challenging. Failing to do so usually causes various unexpected results. In fact, register related bugs have been reported on a regular basis.

During our study, we discover that these features and bugs can help us determine the underlying hypervisor kernel version. A more detailed description of our design approach is presented in Section 3.

2.2 Intel VT-x Extension

As a hardware assisted hypervisor, KVM relies on the virtualization extensions of the underlying processors. In 2006, both Intel (VT-x) and AMD (AMD-SVM) introduced hardware virtualization extensions in their x86 processors. According to their respective manuals, these two technologies are very similar to each other. Because our current implementation of Hyperprobe is based on the Intel processors, we will briefly describe Intel VT-x.

The key concept of Intel VT-x is that the CPU is split into the root mode and the non-root mode. Generally, the hypervisor runs in the root mode and its guests run in the non-root mode. Transitions from the root mode to the non-root mode are called VM entries, and transitions from the non-root mode to the root mode are called VM exits. The hypervisor can specify which instructions and events cause VM exits. These VM exits actually allow the hypervisor to retain control of the underlying physical resources. An example of a VM exit is, when a guest OS attempts to access sensitive registers, such as control registers or debug registers, it would cause a VM exit. A handler defined by the hypervisor will then be invoked, and the hypervisor will try to emulate the behavior of the registers. As mentioned above, given the large number of registers, register emulation is hard and error-prone.

The first generation of Intel VT-x processors mainly simplifies the design of hypervisors. But since then, more and more features have been included in their later processor models. To name a few, Extended Page Table (EPT), which aims to reduce the overhead of address translation, is introduced by Intel since Nehalem processors, and VMCS shadow, which aims to accelerate nested virtualization, is introduced since Haswell. Once these new hardware features are released, modern hypervisors such as KVM and Xen, provide their support for these new features on the software side.

3 Design

Hyperprobe framework has the following goals:

- Practical: The framework should detect the underlying hypervisor kernel version within a reasonable amount of time with high accuracy and precision. As more test cases are added to provide more vantage points of different kernel versions, its accuracy and precision should also be improved.
- Automatic: The framework should run test cases, collect and analyze results automatically without manual intervention. To this end, the test cases should not crash the guest or host OS.[2]
- Extensible: The framework should be easily extended to detect/infer future Linux kernel versions and to add more vantage points to previously released kernel versions. To this end, the whole framework should be modular, and adding modules to the framework should be easy.[3]

3.1 Technical Challenges

To meet these design goals, we faced several challenges: even though the hypervisor introduces new features frequently, how many of them are detectable from within the guest OS? Similarly, how many hypervisor bugs are detectable from within the guest OS?

After manually analyzing the aforementioned 3485 patches, we found a sufficient number of features and bugs that meet our requirements. Tables 1 and 2 illustrate the features and bugs we have selected for our framework. To exploit each, it would require an in-depth knowledge of the kernel and also a good understanding of the particular feature/bug. Due to limited space, we are not able to explain each of the features/bugs, but we will choose some of the more interesting ones and explain them in the next section as case studies. In this section, we elaborate on how we use these features and bugs to infer the underlying hypervisor kernel version.

3.2 KVM Features

KVM releases new features regularly. One may infer the underlying hypervisor kernel version using the following

[2] Kernel bugs that cause guest or host OS to crash are very common, but we purposely avoided using them in our test cases. One could utilize these bugs to gain more vantage points, but they should be used with great caution.

[3] We plan to make Hyperprobe an open source project so that everyone can contribute, making it more robust and accurate.

Table 1: Features We Use in Current Implementation of Hyperprobe

Kernel Major Version	Features	Description
2.6.20		KVM first merged into Linux mainline kernel
2.6.21	Support MSR_KVM_API_MAGIC	Custom MSR register support
2.6.23	SMP support	Support multiple processors for guest OS
2.6.25	Expose KVM CPUID to guest	KVM_CPUID_SIGNATURE
2.6.26	EPT/NPT support	Extended/Nested Page Table
2.6.27	MTRR support	Support the memory type range registers for guest OS
2.6.30	Debug register virtualization	Add support for guest debug
2.6.31	POPCNT support	Support POPCNT instruction in guest OS
2.6.32	KSM support	Kernel Same Page Merging
2.6.34	RDTSCP support, Microsoft Enlightenment	Support RDTSCP instruction and Microsoft Enlightenment
2.6.35	New kvmclock interface	Support paravirtualized clock for the guest
2.6.38	Support MSR_KVM_ASYNC_PF_EN	Enable asynchronous page faults delivery
3.1	Add ”steal time” guest/host interface	Enable steal time
3.2	Support HV_X64_MSR_APIC_ASSIST_PAGE	Support for Hyper-V lazy EOI processing
3.3	PMU v2 support	Expose a version 2 of Performance Monitor Units to guest
3.6	Support MSR_KVM_PV_EOI_EN	Support End of Interrupt Paravirtualization
3.10	Support preemption timer for guest	Support preemption timer for guest
3.12	Nested EPT	Expose Nested Extended Page Table to guest OS
3.13	Support Nested EPT 2MB pages	Expose 2MB EPT page to guest
3.14	Support HV_X64_MSR_TIME_REF_COUNT	Support for Hyper-V reference time counter

Table 2: Bugs We Use in Current Implementation of Hyperprobe

Fixed	Bug Description	Intro'ed
2.6.22	MSR_IA32_MCG_STATUS not writable	2.6.20
2.6.23	MSR_IA32_EBL_CR_POWERON not readable	2.6.20
2.6.25	MSR_IA32_MCG_CTL not readable	2.6.20
2.6.26	MSR_IA32_PERF_STATUS wrong return value upon read	2.6.20
2.6.28	MSR_IA32_MC0_MISC+20 not readable	2.6.20
2.6.30	MSR_VM_HSAVE_PA not readable	2.6.20
2.6.31	MSR_K7_EVNTSEL0 not readable	2.6.20
2.6.32	DR register unchecked access	2.6.20
2.6.34	No support for clear bit 10 of msr register MSR_IA32_MC0_CTL	2.6.20
2.6.35	No support for write 0x100 to MSR_K7_HWCR	2.6.20
2.6.37	MSR_EBC_FREQUENCY_ID not readable	2.6.20
2.6.39	MSR_IA32_BBL_CR_CTL3 not readable	2.6.20
3.2	MSR_IA32_UCODE_REV returns invalid value upon read	2.6.20
3.4	Write 0x8 to MSR_K7_HWCR is buggy	2.6.20
3.5	CPUID returns incorrect value for KVM leaf 0x4000000	2.6.25
3.8	MSR_IA32_TSC_ADJUST not readable	2.6.20
3.9	MSR_AMD64_BU_CFG2 not readable	2.6.20
3.10	MSR_IA32_VMX_ENTRY_CTLS is not set properly as per spec	3.1
3.12	MSR_IA32_FEATURE_CONTROL behave weirdly	3.1
3.14	MSR_IA32_APICBASE reserve bit is writable	2.6.20

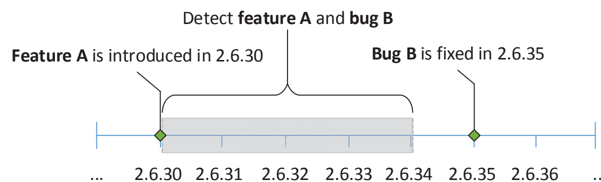


Figure 2: An Inferring Example of The Hyperprobe

logic: if feature *A* is introduced in 2.6.30 and feature *B* is introduced in 2.6.35, then if one can detect feature *A* but not *B*, one may infer that the underlying host kernel version is between 2.6.30 and 2.6.34. However, this may lead to inaccuracies. Since even if feature B is introduced into the Linux mainline kernel on a particular release, the feature could be disabled by system administrators. Therefore, even if feature *B* is not detected, it does not mean the underlying hypervisor kernel version is older than 2.6.35. Such customizations could impact precision.

To avoid such inaccuracies, Hyperprobe uses the following strategy to handle the existence or non-existence of a kernel feature: if we detect a feature exists, we assert that the underlying hypervisor kernel version is no older than the version in which this feature was first introduced. By designing test cases that detect these features, we report a minimum version number. *This number can be viewed as the lower bound of the underlying hypervisor kernel version.*

3.3 KVM Bugs and Bug Fixes

KVM has bugs and bug fixes like any other software. If bugs can be detected from within the guest OS, then one may infer the underlying hypervisor kernel version using the following logic: assuming bug *A* is fixed in kernel version 2.6.30, and bug *B* is fixed in kernel version 2.6.35. If one detects that bug *A* does not exist but bug *B* does, one may infer that the underlying hypervisor kernel is between 2.6.30 and 2.6.34. Similarly, this may lead to inaccuracies, as a bug could be manually fixed in an older kernel without updating the entire kernel. Therefore, the non-existence of a bug does not necessarily mean the kernel is newer than a particular version.

To avoid such inaccuracies, Hyperprobe uses the following strategy to handle the existence or non-existence of a kernel bug: if a bug is detected, we assert that the underlying kernel version is older than the kernel version where this bug is fixed. By creating test cases that detect kernel bugs, we report a maximum version num-

ber. *This number can be viewed as the upper bound of the underlying hypervisor kernel version*. Along with the test cases that detect kernel features, which can report a lower bound, we can then narrow down the hypervisor kernel to a range of versions. Figure 2 illustrates an example: upon the detection of feature A and bug B, we report that the hypervisor has kernel version 2.6.30 as the lower bound and 2.6.34 as the upper bound.

4 Implementation

Our framework implementation consists of 3530 lines of C code (including comments). To meet the extensible goal, we implement the framework of Hyperprobe in a very modular fashion. More specifically, we design 19 test cases for feature detection and 20 test cases for bug detection. Each test case is designed for detecting a specific feature or bug, and is therefore independent of any other test cases. On average, each test case consists of 80 lines of C code. Such a design model makes Hyperprobe fairly extensible. If we identify any other detectable features or bugs later, they can be easily added.

We define two linked lists, named *kvm_feature_testers* and *kvm_bug_testers*. The former includes all the feature test cases, and the latter includes all the bug test cases. Each feature test case corresponds to a kernel version number, which represents the kernel in which the feature is introduced. The feature test cases are sorted using this number and the bug test cases are organized similarly.

Hyperprobe executes as follows. The detection algorithm involves two steps. First, we call the feature test cases in a descending order. As soon as a feature test case returns true, which suggests the feature exists, we stop the loop and report the corresponding number as the lower bound. Second, we call the bug test cases in an ascending order. As soon as a bug test case returns true, which suggests the bug exists, we stop the loop and report the corresponding number as the upper bound.

Most hypervisor kernel features and bugs that we have chosen in this study can be easily detected within the guest OS. In what follows, we describe some of the more interesting ones as case studies.

4.1 Case Studies: Kernel Features

4.1.1 Kernel Samepage Merging

Kernel samepage merging (KSM) [15], introduced in Linux kernel 2.6.32, is a mechanism to save memory, allowing memory overcommitment. This is a crucial feature in a virtualized environment, where there could be a large number of similar VMs running on top of one hypervisor. Other popular hypervisors, such as Xen and VMware, have also implemented similar features [49, 27]. Consequently, if we can detect KSM is enabled we can ascertain that the underlying hypervisor kernel is newer than or equal to version 2.6.32.

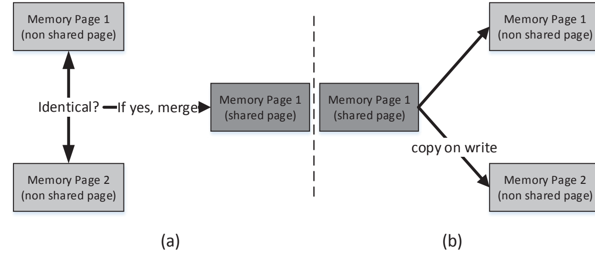


Figure 3: Kernel Same Page Merging
(a) merging identical pages (b) a copy-on-write technique is used when a shared page is modified

KSM scans memory pages and merges those that are identical. Merged pages are set to be copy-on-write, illustrated in Figure 3. This technique is widely used, and it has been proven to be effective in saving memory. However, due to copy-on-write, a write to a shared page incurs more time than a write to a non-shared page. Existing research [46, 50] has shown that this timing difference is large enough to tell if KSM is enabled.

Algorithm 1 describes the procedure of testing KSM. The basic idea of this algorithm is as follows. We first load a random file into memory and write to each page of this file (in memory), then we record the accumulated write access time and call this time *t1*. Next, we load this file again into two separate memory regions, and wait for some time. If KSM is enabled, the identical pages between these two files will be merged. We then write into each page of this file (in memory), and record the accumulated write access time as *t2*. If *t2* is significantly larger than *t1*, namely, the ratio t2/t1 is greater than a predefined threshold, we assume KSM is enabled; otherwise, we assume it is not enabled. In fact, in our testbed, we observe that *t2* is as much as 10 times larger than *t1*. Even in five real cloud environments, we observe that *t2* is still 2 to 5 times larger than *t1*. Thus, we choose 2 as the threshold to detect if KSM is enabled or not.

4.1.2 Extended Page Table (EPT)

Traditionally, commercial hypervisors including KVM, Xen, and VMware, all use the shadow page table technique to manage VM memory. The shadow page table is maintained by the hypervisor and stores the mapping between guest virtual address and machine address. This mechanism requires a serious synchronization effort to make the shadow page table consistent with the guest page table. In particular, when a workload in the guest OS requires frequent updates to the guest page tables, this synchronization overhead can cause very poor performance. To address this problem, recent architecture evolution in x86 processors presents the extended/nested page table technology (Intel EPT and AMD NPT). With this new technology, hypervisors do not need to main-

Algorithm 1: Detecting KSM

```
   Global Var: file
 1 Procedure test_ksm()
 2     load_file_once_into_memory (file);
       // record the clock time before we write
          to each page of the file
 3     time1 ← clock_gettime();
 4     foreach page of file in memory do
 5         write to that page;
       // record the clock time before we write
          to each page of the file
 6     time2 ← clock_gettime();
 7     t1 ← diff(time1,time2);
 8     load_file_twice_into_memory (file);
       // sleep and hope the two copies will be
          merged
 9     sleep (NUM_OF_SECONDS);
       // record the clock time before we write
          to each page of the file
10     time1 ← clock_gettime();
11     foreach page of file in memory do
12         write to that page;
       // record the clock time after we write
          to each page of the file
13     time2 ← clock_gettime();
14     t2 ← diff(time1,time2);
15     ratio ← t2/t1;
16     if ratio > KSM_THRESHOLD then
17         return 1;
18     else
19         return 0;
```

tain shadow page tables for the VMs, and hence avoid the synchronization costs of the shadow page table scenario. The difference between shadow page table and extended page table is illustrated in Figure 4.

Before kernel 2.6.26, KVM uses shadow page table to virtualize memory. Since kernel 2.6.26, KVM starts to support Intel EPT and enable it by default. Therefore, if we can detect the existence of EPT from within the guest OS, we can assume the underlying hypervisor kernel is newer than or equal to version 2.6.26. Algorithm 2 describes the EPT detection mechanism, and we derive this algorithm from the following observations:

- On a specific VM, no matter whether the underlying hypervisor is using shadow page table or EPT, the average time to access one byte in memory is very stable. We have measured this across 30 virtual machines (with different hardware and software configurations). Note that although the time cost may vary across different machines, it remains nearly the same when we switch from EPT to shadow page table, or from shadow page table to EPT.
- When running a benchmark that requires frequent memory mapping changes, EPT offers significant performance improvements over shadow page table. Particularly, we choose the classic forkwait microbenchmark, which has been widely employed [12, 13, 17],to evaluate virtualization performance. The main part of this benchmark repeats the operation of process creation and destruction very aggressively. Similar to [17], we have tested the forkwait microbenchmark across 30 VMs (with different hardware and software configurations), and have consistently observed that EPT offers approximately 600% performance gains over shadow page table.

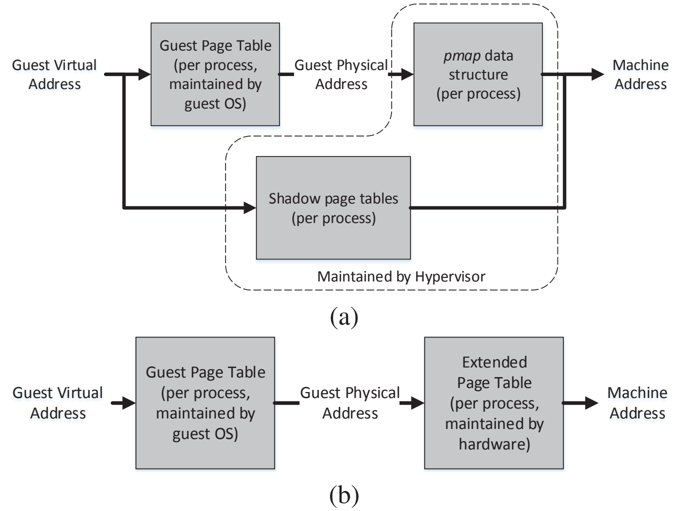


Figure 4: Shadow Page Table and Extended Page Table (a) shadow page table (b) extended page table

Therefore, our algorithm can be elaborated as follows. First we allocate a memory page, compute the average time to access one byte of the memory page, and use this average time as a baseline. Next, we run the forkwait microbenchmark, compute the average time to fork-wait one process, and record the ratio between these two average times (average time to fork-wait one process divided by average time to access one byte of memory page). On all VMs we have tested, this ratio is larger than 100,000 when the hypervisor is using shadow page table, and it is usually between 10,000 to 20,000 when the hypervisor is using EPT. Therefore, we can choose a threshold, and if the ratio is less than that threshold, we assume the underlying hypervisor is using EPT; otherwise, we assume it is using shadow page table. Our current implementation uses 30,000 as the threshold.

4.1.3 Emulating Hyper-V and Support Microsoft Enlightenment

Microsoft Enlightenment is an optimization made by Microsoft to Windows systems when running in a virtualized environment. The key idea is to let the guest OS be aware of the virtualized environment, and therefore tune its behavior for performance improvement. Recent Windows systems, such as Windows Server 2008, and Windows Vista, are fully enlightened [45, 43], which means

Algorithm 2: Detecting EPT

```
Global Var: forkwait_one_process_avg, access_one_byte_avg
1  Procedure forkwait_one_process ()
       // read time stamp counter before we run
          the forkwait benchmark
2      counter1 ← rdtsc();
3      for i ← 0 to NUM_OF_PROCESS do
4          pid ← fork();
5          if pid = 0 then                    // child process
6              exit (0);
7          else
               // parent process, wait until
                  child process exits
8              wait (&status);
       // read time stamp counter when the
          forkwait benchmark is finished
9      counter2 ← rdtsc();
10     cycles ← counter2 − counter1;
       // compute average time for fork-waiting
          one process
11     forkwait_one_process_avg ← cycles/NUM_OF_PROCESS;
12 Procedure access_one_byte (iterations)
13     offset ← 0;
14     page ← malloc(sizeof(PAGE_SIZE));
       // read time stamp counter before we
          access memory bytes
15     counter1 ← rdtsc();
16     for i ← 0 to iterations do
17         page[offset] ← (page[offset] + 1)  mod 256;
18         offset ← (offset + 1)  mod PAGE_SIZE;
       // read time stamp counter after we
          access memory bytes
19     counter2 ← rdtsc();
20     cycles ← counter2 − counter1;
       // compute average time for accessing
          one byte
21     access_one_byte_avg ← cycles/iterations;
22 Procedure one_time_run()
23     access_one_byte(num_of_iterations);
24     forkwait_one_process();
25     ratio ← forkwait_one_process_avg/access_one_byte_avg;
26     if ratio < EPT_THRESHOLD then
27         return 1;
28     else
29         return 0;
30 Procedure test_ept()
31     for i ← 0 to LOOP_NUMBER do
32         if one_time_run() = 1 then
33             return 1 ;
34     return 0;
```

they take full advantage of the possible enlightenments.

Microsoft Enlightenment was originally designed for Hyper-V, but Microsoft provides APIs for other hypervisors to utilize this optimization. Since kernel 2.6.34, KVM has started utilizing these APIs and supporting Microsoft Enlightenment. According to the Hyper-V specification [6, 7], several synthetic registers are defined, including HV_X64_GUEST_OS_ID, HV_X64_HYPERCALL, HV_X64_VP_INDEX, as well as the EOI/TPR/ICR APIC registers. Details of these registers are shown in Table 3. Before kernel 2.6.34, accessing these registers would generate a general protection fault, but since kernel 2.6.34, they should be accessible whether accessing from a Windows or Linux guest OS. Thus, we attempt to access these registers. If they are accessible, we assume the kernel version is newer than or equal to version 2.6.34; otherwise, the feature may not be present, but we do not make any assertion regarding the underlying kernel version. In addition, in some later kernel versions, more Hyper-V defined synthetic registers are emulated by KVM. For example, HV_X64_MSR_TIME_REF_COUNT is emulated in kernel 3.14. Thus, successful access to the register suggests that the underlying hypervisor kernel should be as new as 3.14.

4.2 Case Studies: Kernel Bugs

4.2.1 Debug Register Unchecked Access

Debug registers are protected registers. They should only be accessed by ring 0 code, namely kernel code. However, before kernel 2.6.32, KVM does not check the privilege of the guest code that accesses the debug registers. Therefore, any process, regardless of its current privilege level (CPL), is able to read from and write to debug registers. This leads to a security issue in the guest OS, as attackers might be able to implement the infamous DR rootkit [14, 28] without installing a kernel module, thus making the rootkit more difficult to detect even from the hypervisor level.

On kernel 2.6.32, KVM maintainer, Avi Kivity, submitted a patch that fixed this bug. The patch would check the CPL before accessing debug registers, and would generate a fault if the CPL is greater than zero. We built a simple test case based on this bug. The basic idea is to use the fork system call to create a child process, and let the child process try to access a debug register. If the bug is fixed, the child process should be terminated by a segmentation fault signal. But if the bug has not yet been fixed, the child process will continue to run and eventually exit normally. Therefore, we let the parent process wait until the child process exits, and check the exit status of the child process. If it exits normally, we report the bug still exists; otherwise, we report the bug is fixed.

4.2.2 Model Specific Register (MSR) Bugs

CPU vendors such as Intel and AMD define hundreds of model specific registers on their processors. Some of these registers are common across different types of processors, while others might only exist in a specific processor. Due to the large variety of such registers, over the years, emulating the behavior of these registers has always been a painful task in modern hypervisors. Because of this, Andi Kleen, a key maintainer of Linux kernels, who used to be in charge of the x86_64 and i386 architectures, believes that it is impossible to emulate a

Table 3: Hyper-V Defined Synthetic Registers

Register Name	Address	Description	Supported in Linux Kernel Since
HV_X64_MSR_GUEST_OS_ID	0x40000000	Used to identify guest OS	2.6.34
HV_X64_MSR_HYPERCALL	0x40000001	Used to enable/disable Hypercall	2.6.34
HV_X64_MSR_VP_INDEX	0x40000002	Used to identify virtual processor	2.6.34
HV_X64_MSR_EOI	0x40000070	Fast access to APIC EOI register	2.6.34
HV_X64_MSR_ICR	0x40000071	Fast access to APIC ICR register	2.6.34
HV_X64_MSR_TPR	0x40000072	Fast access to APIC TPR register	2.6.34
HV_X64_MSR_APIC_ASSIST_PAGE	0x40000073	Used to enable/disable lazy EOI processing	3.2
HV_X64_MSR_TIME_REF_COUNT	0x40000020	Time reference counter	3.14

particular CPU 100% correctly [33].

However, incorrect emulation of these registers could cause problems in the guest OS. For example, to fix their hardware defects, Intel defines a capability in its Pentium 4, Intel Xeon, and P6 family processors called microcode update facility. This allows microcode to be updated if needed to fix critical defects. After microcode is updated, its revision number is also updated. BIOS or OS can extract this revision number via reading the MSR register IA32_UCODE_REV, whose address is 0x8BH. Previously, in Linux kernel, when the guest tries to read this register, KVM would return an invalid value, which is 0, and this would cause Microsoft Windows 2008 SP2 server to exhibit the blue screen of death (BSOD). To fix this problem, since kernel 3.2, KVM reports a non-zero value when reading from IA32_UCODE_REV. Details of this bug fix can be found in [47].

Our detection is also straightforward: Linux kernel provides a kernel module called *msr* that exports an interface through file /dev/cpu/cpuN/msr, where N refers to the CPU number. This interface allows a user level program to access MSR registers. Therefore, we can detect the bug by accessing this file with the address of IA32_UCDOE_REV, which is 0x0000008b according to Intel's manual. If a read to this register returns 0, we can assert that the bug exists.

5 Evaluation

To demonstrate how Hyperprobe performs in the wild, we ran its test suite on VMs provisioned from different public cloud providers to detect their hypervisor kernel versions. In most cases, we were able to narrow the suspected hypervisor kernel versions down to a few; in one case, we even had an exact match. However, as public cloud providers do not disclose detailed information about the hypervisors they are using (for obvious security reasons), we had to find other means to confirm these results, such as user forums and white papers. Our results do coincide with what are being reported via these side channels. To more rigorously verify the accuracy of Hyperprobe, we also evaluated it in a controlled testbed environment across 35 different kernel versions with very encouraging results.

5.1 Results in Real World Clouds

The public cloud providers we selected in this study include Google Compute Engine, HP Helion Public Cloud, ElasticHosts, Joyent Cloud, and CloudSigma. (all KVM-based) In our experiments, we intentionally created VMs with different configurations to test the detection robustness and accuracy of our framework. The results are shown in Tables 4, 5, 6, 7, and 8. Running the test suite and analyzing the collected results take less than 5 minutes to complete, which is fairly reasonable from a practical point of view. In fact, we observe that the running time is mainly dominated by those test cases that require sleeping or running some microbenchmarks. In what follows, we detail our findings for each cloud provider.

5.1.1 Google Compute Engine

Google Compute Engine is hosted in data centers located in Asia, Europe, and America. One can choose the number of VCPUs per VM ranging from 1 to 16. Hyperprobe shows that Google is using a kernel version between 3.2 and 3.3 in its hypervisors. According to a recent work [37] and some online communications written by Google engineers [32, 1], Debian 7 is most likely used in its hypervisors as this Linux distribution is widely used in its production environments. The default kernel of Debian 7 is 3.2.0-4, agreeing with our findings.

5.1.2 HP Helion Public Cloud

HP Helion Public Cloud is hosted in data centers in U.S. East and West regions. One can choose the number of VCPUs per VM ranging from 1 to 4. Hyperprobe detected that the HP cloud is using a kernel version between 3.2 and 3.7 in its hypervisors. According to some unofficial online documents and web pages [4, 5], HP is most likely using Ubuntu 12.04 LTS server as its host OS. The default kernel of Ubuntu 12.04 LTS is 3.2, falling within the range reported by our framework.

5.1.3 ElasticHosts

ElasticHosts is the first public cloud service provider to use Linux-KVM as its hypervisors [2]. Its data centers are located in Los Angeles, CA and San Antonio, TX. For free trial users, a VM with only 1 VCPU and 1GB of memory is given. Hyperprobe reported that the underlying hypervisor kernel version should be 3.6 to 3.8.

Table 4: Inferring Host Kernel Version in Google Compute Engine

VM Name	Zone	Machine Type	Image	VCPU	VCPU Frequency	RAM	Disk	Min	Max
gg-test1	asia-east1-a	n1-standard-1	SUSE SLES 11 SP3	1	2.50GHZ	3.8GB	10G	3.2	3.3
gg-test2	asia-east1-b	n1-highcpu-16	SUSE SLES 11 SP3	16	2.50GHZ	14.4GB	10G	3.2	3.3
gg-test3	us-central1-a	n1-highmem-16	Debian 7 wheezy	16	2.60GHZ	8GB	104G	3.2	3.3
gg-test4	us-central1-b	f1-micro	backports Debian 7 wheezy	1	2.60GHZ	4GB	0.6G	3.2	3.3
gg-test5	europe-west1-a	n1-highmem-4	backports Debian 7 wheezy	4	2.60GHZ	26GB	10G	3.2	3.3
gg-test6	europe-west1-b	n1-standard-4	Debian 7 wheezy	4	2.60GHZ	15GB	10G	3.2	3.3

Table 5: Inferring Host Kernel Version in HP Helion Cloud (3 Month Free Trial)

VM Name	Region	Zone	Size	Image	VCPU	VCPU Frequency	RAM	Disk	Min	Max
hp-test1	US East	az2	standard xsmall	SUSE SLES 11 SP3	1	2.4GHZ	1GB	20G	3.2	3.7
hp-test2	US East	az2	standard xlarge	SUSE SLES 11 SP3	4	2.4GHZ	15GB	300G	3.2	3.7
hp-test3	US East	az3	standard large	SUSE SLES 11 SP3	4	2.4GHZ	8GB	160G	3.2	3.7
hp-test4	US East	az1	standard medium	SUSE SLES 11 SP3	2	2.4GHZ	4GB	80G	3.2	3.7
hp-test5	US West	az1	standard medium	Ubuntu 10.04	2	2.4GHZ	4GB	80G	3.2	3.7
hp-test6	US West	az3	standard xlarge	Debian Wheezy 7	4	2.4GHZ	15GB	300G	3.2	3.7

Table 6: Inferring Host Kernel Version in ElasticHosts Cloud (5 Day Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
eh-test1	Los Angeles	Ubuntu 13.10	2.8GHz	1GB	10GB	3.6	3.8
eh-test2	Los Angeles	Cent OS Linux 6.5	2.8GHz	512MB	5GB SSD	3.6	3.8
eh-test3	Los Angeles	Debian Linux 7.4	2.8GHz	512MB	5GB	3.6	3.8
eh-test4	Los Angeles	Ubuntu 14.04 LTS	2.8GHz	1GB	10GB	3.6	3.8
eh-test5	San Antonio	Ubuntu 12.04.1 LTS	2.5GHz	1GB	5GB	3.6	3.8
eh-test6	San Antonio	CentOS Linux 6.5	2.5GHz	512MB	10GB	3.6	3.8

For this provider, we were not able to find information to confirm if our finding is correct.

5.1.4 Joyent Cloud

Joyent Cloud is yet another IaaS cloud service provider that uses KVM as its hypervisors [11]. Its data centers are located in U.S. East, West, and Southwest regions, as well as in Amsterdam, Netherlands. It provides a one-year free trial with very limited resources (i.e., 0.125 VCPU and 256MB of memory). Hyperprobe reported that the hypervisors hosting the free trial machines are using a rather old 2.6.34 kernel (an exact match).

Further investigation showed that Joyent runs a custom kernel called SmartOS in its hypervisors. It was created based on Open Solaris and Linux KVM, and we confirmed that Linux 2.6.34 is the version that Joyent engineers have ported into SmartOS [18].

5.1.5 CloudSigma

CloudSigma is an IaaS cloud service provider based in Zurich, Switzerland. However, its data centers are located in Washington, D.C. and Las Vegas, NV. For free trial users, only one VCPU with 2GB of memory can be obtained. Hyperprobe reported that the underlying hypervisor kernel version should be between 3.6 and 3.13.

The main reason that CloudSigma's result spans a wider range than others is its usage of AMD processors in its data centers. Our current implementation of Hyperprobe is optimized only for Intel processors. KVM includes an architecture dependent module, namely *kvm-intel.ko* and *kvm-amd.ko*, for Intel and AMD, respectively. Although some features and bugs are common in both architectures, others may not be. And these architecture-specific features and bugs can further improve the accuracy of Hyperprobe's reported results. The result for CloudSigma was mainly based on the common features and bugs, and thus, Hyperprobe was not able to narrow down the kernel versions as much as it could for the Intel-based cloud providers.

5.1.6 Summarizing Findings in Public Clouds

We found several interesting facts about these clouds:

- Even if a cloud provider has multiple data centers spread across various geographic locations, it is very likely that they are using the same kernel version and distribution. This confirms the conventional wisdom that standardization and automation are critical to the maintainability of an IT environment as it grows more complex. Modern cloud providers' data centers are as complicated as they can get.
- Cloud providers usually do not use the latest kernel. At the time of our study, the latest stable Linux kernel is version 3.14, which was released in March 2014, and our experiments were performed in June 2014. However, we can see cloud providers like HP and ElasticHosts are still using kernels older than version 3.8, which was released in February 2013. Google and Joyent Cloud are using even older kernels. This is understandable as newer kernels might not have been extensively tested, and therefore, it could be risky to use them for production workloads.

5.2 Results in a Controlled Testbed

To better observe if what Hyperprobe detects is really what is deployed, we ran the same test suite in a con-

Table 7: Inferring Host Kernel Version in Joyent Cloud (1 Year Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
jy-test1	US-East	CentOS 6.5	3.07GHz	250MB	16GB	2.6.34	2.6.34
jy-test2	US-SouthWest	Ubuntu Certified 14.04	2.40GHz	250MB	16GB	2.6.34	2.6.34
jy-test3	US-West	CentOS 6.5	2.40GHz	250MB	16GB	2.6.34	2.6.34
jy-test4	EU-Amsterdam	Ubuntu Certified 14.04	2.40GHz	250MB	16GB	2.6.34	2.6.34

Table 8: Inferring Host Kernel Version in CloudSigma (7 Day Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
cs-test1	Washington DC	CentOS 6.5 Server	2.5GHz	2GB	10GB SSD	3.6	3.13
cs-test2	Washington DC	Fedora 20 Desktop	2.5GHz	1GB	10GB SSD	3.6	3.13
cs-test3	Washington DC	Debian 7.3 Server	2.5GHz	512MB	10GB SSD	3.6	3.13
cs-test4	Washington DC	SUSE SLES 11 SP3	2.5GHz	2GB	10GB SSD	3.6	3.13

trolled testbed environment across all the 35 major Linux kernel releases (2.6.20 to 3.14) since KVM was first introduced. The testbed is a Dell Desktop (with Intel Xeon 2.93GHz Quad-Core CPU and 2GB memory) running OpenSuSE 11.4. We used OpenSuSE 11.4 as the guest OS running a 3.14 Linux kernel. We manually compiled each of the 35 kernels and deployed it as the kernel used in our hypervisor. After each set of experiments, we shut down the guest OS and rebooted the host OS.

The results are listed in Table 9. To sum up, from Table 9, it can be seen that, among the 35 host OS kernel versions, we can find an exact match for 11 of them; for 15 of them, we can narrow down to 2 versions; for 4 of them, we can narrow down to 3 versions; for 4 of them, we can narrow down to 4 versions; and for 1 of them, we can narrow down to 5 versions.

Table 9: Inferring Results in a Controlled Testbed

Kernel	Reported Min	Reported Max	Accuracy
2.6.20	2.6.20	2.6.21	2 versions
2.6.21	2.6.21	2.6.21	exact match
2.6.22	2.6.21	2.6.22	2 versions
2.6.23	2.6.23	2.6.24	2 versions
2.6.24	2.6.23	2.6.24	2 versions
2.6.25	2.6.25	2.6.25	exact match
2.6.26	2.6.25	2.6.27	3 versions
2.6.27	2.6.27	2.6.27	exact match
2.6.28	2.6.27	2.6.29	3 versions
2.6.29	2.6.27	2.6.29	3 versions
2.6.30	2.6.30	2.6.30	exact match
2.6.31	2.6.31	2.6.31	exact match
2.6.32	2.6.32	2.6.33	2 versions
2.6.33	2.6.32	2.6.33	2 versions
2.6.34	2.6.34	2.6.34	exact match
2.6.35	2.6.35	2.6.36	2 versions
2.6.36	2.6.35	2.6.36	2 versions
2.6.37	2.6.35	2.6.38	4 versions
2.6.38	2.6.38	2.6.38	exact match
2.6.39	2.6.38	3.1	4 versions
3.0	2.6.38	3.1	4 versions
3.1	3.1	3.1	exact match
3.2	3.2	3.3	2 versions
3.3	3.3	3.3	exact match
3.4	3.3	3.4	2 versions
3.5	3.3	3.7	5 versions
3.6	3.6	3.7	2 versions
3.7	3.6	3.7	2 versions
3.8	3.6	3.8	3 versions
3.9	3.6	3.9	4 versions
3.10	3.10	3.11	2 versions
3.11	3.10	3.11	2 versions
3.12	3.12	3.13	2 versions
3.13	3.13	3.13	exact match
3.14	3.14	3.14	exact match

6 Discussion

In this section, we discuss some potential enhancements.

6.1 Other Hypervisors

Our current framework is developed for KVM, but the approach we propose should certainly work for other popular hypervisors such as Xen. In fact, we notice that KVM and Xen share many of the same features and bugs. For instance, They both support the Microsoft enlightenment feature, and we also notice that some MSR register bugs exist in both KVM and Xen. Therefore, we plan to include the support for Xen hypervisors in our framework.

Meanwhile, we are also trying to enhance our framework for closed-source hypervisors, such as VMware and Hyper-V. Even though their source codes are not available, the vendors provide a release note for each major release, which clearly states their new features. And the bugs of these hypervisors are also publicly available.

6.2 Open Source

We have implemented Hyperprobe as a framework, which includes different test cases, but each test case is totally separated from all the other test cases. In other words, each test case can be developed separately. Such a key property allows it to meet one of our design goals: extensible. In fact, we plan to make it open source, so that we can rely on a community of users to use it and contribute additional test cases. The more vantage points (i.e., test cases) we have, the better precision our detection result can achieve. And this will certainly accelerate our development process and our support for the other hypervisors.

7 Related Work

We survey related work in two categories: detection of a specific hypervisor and attacks against hypervisors.

7.1 Detection of Hypervisors

Since virtualization has been widely used for deploying defensive solutions, it is critical for attackers to be able to detect virtualization, i.e., detect the existence of a hypervisor. To this end, several approaches have been proposed for detecting the underlying hypervisors and are briefly described as follows.

RedPill [44] and Scooby Doo [34] are two techniques proposed to detect VMware, and they both work because VMware relocates some sensitive data structures such as Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT), and Local Descriptor Table (LDT). Therefore, one can examine the value of the IDT base, if it exceeds a certain value or equals a specific hard-coded value, then one assumes that VMware is being used. However, these two techniques are both limited to VMware detection and are not reliable on machines with multi-cores [41]. By contrast, the detection technique proposed in [41] is more reliable but only works on Windows guest OSes. Their key observation is that because LDT is not used by Windows, the LDT base would be zero in a conventional Windows system but non-zero in a virtual machine environment. Therefore, one can simply check for a non-zero LDT base on Windows and determine if it is running in VMware environment.

A variety of detection techniques based on timing analysis have also been proposed in [25, 23].The basic idea is that some instructions (e.g., RDMSR) are intercepted by hypervisors and hence their execution time is longer than that on a real machine. One can detect the existence of a hypervisor by measuring the time taken to execute these instructions. Note that all these previous works can only detect the presence of a hypervisor and/or its type, but none are able to retrieve more detailed information about the underlying hypervisor, such as its kernel version.

7.2 Attacks against Hypervisors

Modern hypervisors often have a large code base, and thus, are also prone to bugs and vulnerabilities. Considering a hypervisor's critical role in virtualized environments, it has been a particularly attractive target for attackers. Vulnerabilities in hypervisors have been exploited by attackers, as demonstrated in prior work [35, 21]. Perez-Botero et al. [39] characterized various hypervisor vulnerabilities by analyzing vulnerability databases, including SecurityFocus [10] and NIST's Vulnerability Database [9]. Their observation is that almost every part of a hypervisor could have vulnerabilities. Ormandy [38] classified the security threats against hypervisors into three categories: total compromise, partial compromise, and abnormal termination. A total compromise means a privilege escalation attack from a guest OS to the hypervisor/host. A partial compromise refers to information leakage. An abnormal termination denotes the shut down of a hypervisor caused by attackers. According to the definition above, gaining hypervisor information by Hyperprobe belongs to a partial compromise.

8 Conclusion

In this paper, we investigated the reverse information retrieval problem in a virtualized environment. More specifically, we coined the term virtual machine extrospection (VME) to describe the procedure of retrieving the hypervisor information from within a guest OS. As a first step towards VME, we presented the design and development of the Hyperprobe framework. After analyzing the seven-year evolution of Linux KVM development, including 35 kernel versions and approximately 3485 KVM related patches, we implemented test cases based on 19 hypervisor features and 20 bugs. Hyperprobe is able to detect the underlying hypervisor kernel version in less than five minutes with a high accuracy. To the best of our knowledge, we are the first to study the problem of detecting host OS kernel version from within a VM. Our framework generates promising results in five real clouds, as well as in our own testbed.

References

[1] Bringing debian to google compute engine. http://googleappengine.blogspot.com/2013/05/bringing-debian-to-google-compute-engine_9.html.

[2] Elastichosts wiki page. http://en.wikipedia.org/wiki/ElasticHosts.

[3] Google compute engine. https://cloud.google.com/products/compute-engine/.

[4] Hp cloud os faqs. http://docs.hpcloud.com/cloudos/prepare/faqs/.

[5] Hp cloud os support matrix for hardware and software. http://docs.hpcloud.com/cloudos/prepare/supportmatrix/.

[6] Hypervisor top-level functional specification 2.0a: Windows server 2008 r2. http://www.microsoft.com/en-us/download/details.aspx?id=18673.

[7] Hypervisor top-level functional specification 3.0a: Windows server 2012. http://www.microsoft.com/en-us/download/details.aspx?id=39289.

[8] Joyent. http://www.joyent.com/.

[9] National vulnerability database. http://nvd.nist.gov/.

[10] Security focus. http://www.securityfocus.com/.

[11] Virtualization performance: Zones, kvm, xen. http://dtrace.org/blogs/brendan/2013/01/11/virtualization-performance-zones-kvm-xen/.

[12] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 41*, 11 (2006), 2–13.

[13] AHN, J., JIN, S., AND HUH, J. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)* (2012), IEEE Computer Society, pp. 476–487.

[14] ALBERTS, B. Dr linux 2.6 rootkit released. `http://lwn.net/Articles/296952/`.

[15] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *Proceedings of the Linux Symposium* (2009), pp. 19–28.

[16] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2010), vol. 10, pp. 423–436.

[17] BHATIA, N. Performance evaluation of intel ept hardware assist. *VMware, Inc* (2009).

[18] CANTRILL, B. Experiences porting kvm to smartos. *KVM Forum 2011*.

[19] Cloudsigma. `https://www.cloudsigma.com/`.

[20] Elastichosts. `http://www.elastichosts.com/`.

[21] ELHAGE, N. Virtunoid: Breaking out of kvm. *Black Hat USA* (2011).

[22] FERRIE, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[23] FRANKLIN, J., LUK, M., JONATHAN, M., SESHADRI, A., PERRIG, A., AND VAN DOORN, L. Towards sound detection of virtual machines. *Advances in Information Security, Botnet Detection: Countering the Largest Security Threat*, 89–116.

[24] FRANKLIN, J., LUK, M., MCCUNE, J. M., SESHADRI, A., PERRIG, A., AND VAN DOORN, L. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review 42*, 3 (2008), 83–92.

[25] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of the 9th USENIX workshop on Hot topics in operating systems (HotOS)* (2007).

[26] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security (NDSS)* (2003), pp. 191–206.

[27] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A., VARGHESE, G., VOELKER, G., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Proceedings of the 8th symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[28] HALFDEAD. Mistifying the debugger, ultimate stealthness. `http://phrack.org/issues/65/8.html`.

[29] Hp helion public cloud. `http://www.hpcloud.com/`.

[30] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)* (2007), pp. 128–138.

[31] JIANG, X., AND XU, D. Collapsar: A vm-based architecture for network attack detention center. In *USENIX Security Symposium* (2004), pp. 15–28.

[32] KAPLOWITZ, J. Debian google compute engine kernel improvements, now and future. `https://lists.debian.org/debian-cloud/2013/11/msg00007.html`.

[33] KLEEN, A. Kvm mailing list discussion. `https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg611255.html`.

[34] KLEIN, T. Scooby doo-vmware fingerprint suite. `http://www.trapkit.de/research/vmm/scoopydoo/index.html`, 2003.

[35] KORTCHINSKY, K. Cloudburst: A vmware guest to host escape story. *Black Hat USA* (2009).

[36] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium* (2008), pp. 243–258.

[37] MERLIN, M. Live upgrading thousands of servers from an ancient red hat distribution to 10 year newer debian based one. In *Proceedings of the 27th conference on Large Installation System Administration (LISA)* (2013), pp. 105–114.

[38] ORMANDY, T. An empirical study into the security exposure to hosts of hostile virtualized environments. `http://taviso.decsystem.org/virtsec.pdf`, 2007.

[39] PEREZ-BOTERO, D., SZEFER, J., AND LEE, R. B. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing* (2013), ACM, pp. 3–10.

[40] PETRONI JR, N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)* (2007), pp. 103–115.

[41] QUIST, D., AND SMITH, V. Detecting the presence of virtual machines using the local data table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`, 2006.

[42] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection (RAID)* (2008), Springer, pp. 1–20.

[43] RUSSINOVICH, M. Inside windows server 2008 kernel changes. *Microsoft TechNet Magazine* (2008).

[44] RUTKOWSKA, J. Red pill... or how to detect vmm using (almost) one cpu instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[45] SMYTH, N. *Hyper-V 2008 R2 Essentials*. eBookFrenzy, 2010.

[46] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Software side channel attack on memory deduplication. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 11' POSTER)* (2011).

[47] TOSATTI, M. Kvm: x86: report valid microcode update id. `https://github.com/torvalds/linux/commit/742bc67042e34a9fe1fed0b46e4cb1431a72c4bf`.

[48] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review 39*, 5 (2005), 148–162.

[49] WALDSPURGER, C. Memory resource management in vmware esx server. *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI) 36*, SI (2002), 181–194.

[50] XIAO, J., XU, Z., HUANG, H., AND WANG, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013).

Dynamic Provisioning of Storage Workloads

Jayanta Basak
NetApp Inc.

Madhumita Bharde
NetApp Inc.

Abstract

Due to lack of generic, accurate, dynamic and comprehensive models for performance estimation, customers typically tend to under-provision or over-provision storage systems today. With multi-tenancy, virtualization, scale and unified storage becoming norms in the industry, it is highly desirable to strike an optimum balance between utilization and performance. However, performance prediction for enterprise storage systems is a tricky problem, considering that there are multiple hardware and software layers cascaded in complex ways that affect behavior of the system. Configuration factors such as CPU, cache size, RAM size, capacity, storage backend (HDD/Flash) and network cards etc. are known to have significant effect on the number of IOPS that can be pushed to the system. However, apart from system characteristics as these, storage workloads vary reasonably and therefore, IOPS numbers depend heavily on types of workloads provisioned on storage systems. In this work, we treat storage system as a hybrid of black-box and white-box models, and propose a solution that will enable administrators to make decisions in the presence of multiple workloads dynamically. Our worst-case prediction is within 15% error margin.

1 Introduction

There is immense pressure on storage providers to increase utilization of their resources while maintaining performance guarantees. A storage resource can be operated at 'knee of the curve' - e.g. 70% of resource utilization, as a thumb rule. However, identifying the 'knee of the curve' dynamically is a challenge. The situation becomes more complicated for a mix of different workloads as response times are sensitive to workload characteristics. Moreover, any aggressive provisioning of storage resources can result in performance impact hitting bottom-line for the resource provider. To avoid such situations, storage providers often over-provision storage resources and system remains under-utilized.

Optimum resource utilization is also crucial in cloud provider environment. Usually, cloud providers thin provision resources. They need to be able to seamlessly provision containers, migrate VMs, and redistribute the resource pool among client applications [6, 8, 11, 13, 19, 21, 23, 24]. So it is extremely important to dynamically estimate the actual maximum throughput that can be delivered for these application environments. For example, with white-box like static provisioning, the system is utilized $40-50\%$ whereas the aim is to dynamically provision the workloads such that 70% utilization becomes a realistic goal.

Performance headroom modeling is typically done via two approaches – white-box and black-box. In white-box models [5], each component like CPU, disk, network, and memory is modeled using queueing theory. For each component, queueing delay for a certain IO request is computed and individual models are aggregated to obtain overall response time. Black-box models [7, 9, 12, 25, 26], on the contrary, model the entire system as a black-box and use machine learning techniques to predict the relationship between IO pattern and response time. White-box models are usually static in nature; but are highly tunable in terms of system parameters. On the other hand, black-box models can predict well in the dynamic environments but usually have less control on tuning of system parameters.

In this paper, we take a hybrid approach where we learn the system behavior in terms of characterizing the dependency of response time (latency) with IOPS. We also use concepts from queueing theory to model mixture of workloads on multiple volumes (logical containers) in an aggregate (physical container or set of disks). We leverage the advantages of black-box models for dynamic provisioning and that of white-box models for handling multiple workloads. We predict

the maximum IOPS possible per volume basis in an aggregate on a node. Our aim is to achieve a worst-case error margin of 15% for this prediction. The prediction of maximum IOPS is further extended for many interesting use cases such as:

- What is the maximum number IOPS that can be pushed for an existing workload?
- What would be maximum number of IOPS for a new workload, given its characteristics?
- What is the effect on existing workloads if a new workload is provisioned/migrated?
- In a storage cluster, what would be the best place to provision a workload?
- Can we redistribute workloads in the cluster for optimum utilization and performance?

The rest of the paper is organized as follows. Section 2 surveys the literature around white-box and black-box models for performance of storage systems. Section 3 describes motivations and challenges for a comprehensive storage provisioning solution. Section 4 gives details of techniques we came up with for storage provisioning. Section 5 presents experimental design and results. Finally, Section 6 summarizes and concludes the paper giving a glimpse in the possible future work.

2 Related Work

In this section, we provide a brief survey of the existing literature that is related to our work. White-box approaches for storage system modeling [5, 10] have existed for over three decades and are still being actively investigated in modern day mass storage systems [14]. White box approaches usually model individual components like CPU, memory, disk, and network are and compute IO requests delay as the queuing delay. The individual queuing delays are then aggregated to obtain the overall response time. In computing the overall response time, the workload characteristics are embedded in the sequence of read-write of operations of the IO pattern.

In black-box modeling, various machine learning techniques are applied to model storage system behavior such as the dependency of response time (latency) with IOPS. BASIL [7] and Pesto [9], Relative Fitness [12] and CMU-CART [25, 26] are three black-box techniques that have been proposed for modeling storage systems. BASIL provides predictions in the interpolation (trained) region. Pesto can extrapolate in the unseen region under the assumption that latency has a linear relationship with the outstanding IO. Both in relative fitness [12] and CMUCART [26], the performance of the storage system is predicted based on observed samples from the past. On a similar line, CART [4] has been used in black-box modeling of the storage performance in [28]. In these approaches, certain counters are considered, and a CART model is built to model latency and throughput. Very recently, bagging [3] of the CART models has been used for better prediction of the storage system performance [29].

In machine learning literature [22, 2], function approximation refers to predicting the functional value for an unobserved sample. In support vector regression (SVR) [22, 17] with RBF kernels, the prediction is usually good for the interpolation region. Support vector regression with polynomial kernels can be used for extrapolation. From the storage provisioning perspective, it is essential to associate a confidence level with the prediction to suggest to the user if the prediction is reliable or not. Kriging [27, 16] is able to associate a confidence level with each prediction. With a modification in computation of model variogram from the experimental variogram, Gaussian Process (GP) model has been developed [15] which also associates confidence level with prediction, although GP is suitable for interpolation only. In a recently developed black-box model called M-LISP [1], kriging has been used to successfully extrapolate the storage system performance for unseen amount of workload, and it is able to associate confidence interval with the prediction.

3 Motivation and Problem Statement

In a Latency (response time) vs IOPS curve for a storage system, the response time remains almost constant in low IOPS region even if more IOPS are pushed to the storage system. After the number of IOPS reaches a 'knee' , latency suddenly increases within a short range of IOPS. If IOPS are increased even further, after a point, no more IOPS can be pushed to the system and the latency shoots up drastically. It is desirable to operate at the knee for reasonable balance between performance and utilization. However, it is extremely difficult to theoretically quantify the

'knee' of the curve for a storage system. As a rule of thumb, industry practitioners use 70% of the maximum IOPS as the 'knee of the curve'.

Most of black-box modeling techniques in the literature predict the response time (latency) for certain IOPS based on the interpolation mechanism. Extrapolation techniques predict the response time for certain IOPS in an unseen region..Pesto [9] and M-LISP [1] perform extrapolation to predict the storage system performance. However, it is very difficult to apply them for mixed workload situation. For example, different workloads are typically deployed in different volumes and having $10-20$ volumes is quite common in industrial scenario [1]. For such cases, prediction for a new volume in absence of existing volumes is not applicable in their presence. Secondly, it is important to provision new volumes so that they don't affect performance of existing volumes. So, it's crucial for a customer to know the maximum possible IOPS that the system may provide for the new volume/workload even before it is actually provisioned. Existing black-box approaches estimate the maximum IOPS per volume on a running system and do not estimate that for a new volume. Also existing black-box approaches are not equipped to model multiple workloads due to their interference. It is possible to apply white-box models to appropriately capture the resource constraints but they are not generic and dynamic.

In this paper, we use a black-box model to capture the dependency of the response time with IOPS and predict the maximum IOPS per volume in an aggregate. We correlate the response characteristics with parameters of workloads. We then model the entire black-box server as a multi-queue single-server model to take into account of the multiple workloads running on multiple volumes in the same aggregate. For new workloads, we compute the maximum possible IOPS, given a system configuration, for various different workload characteristics in the laboratory environment before commissioning the system and construct look-up tables for the same. We observed that for a given configuration, the maximum possible IOPS depends on workload characteristics. The maximum IOPS for a given configuration and a given workload type may change due to system upgrade, file-system aging and fragmentation. We designed a provisioning system that takes feedback from the environment and dynamically updates the tables to adapt to these changes. Details are provided in Sections 4 and 5.

[1] As observed from customer systems

4 Details of the Approach

4.1 Maximum IOPS for a Single Workload

We have considered the entire storage system as a simple black-box server serving a queue of IO requests. Outstanding IOs (OIO) is a measure of the queue length (depth). From Little's Law[2], (OIO) can be expressed as

$$OIO = Latency \times IOPS \tag{1}$$

For higher IOPS, Latency is directly proportional to OIO to be served [9]. Therefore, we have

$$Latency = a \times OIO + b \tag{2}$$

where a and b are constants. From Equations (1) and (2), we have

$$Latency = \frac{b}{1 - a \times IOPS} \tag{3}$$

This derivation is also depicted in Figure 1.

As the storage system gets saturated, i.e., the denominator of Equation (3) becomes close to zero, latency value tends infinity (as shown in Figure 1). Substituting in Equation (8), we have

$$Maximum\ IOPS = 1/a \tag{4}$$

Note that, Equation(2) is true asymptotically. For small number of IOPS, the relationship may not hold true; however, we model the system in high IOPS region.

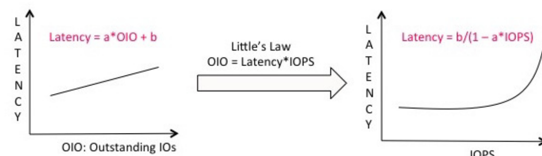


Figure 1: Fundamental Technique

As is evident from the Equation (4), maximum IOPS is the inverse of slope of the line representing the linear relationship between Latency

[2] URL:http://web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf

vs OIO. Note that, this value $1/a$ could also be considered as a service rate of the system for a given workload. This terminology will be revisited later in Section 4.4.

We gathered periodic measurements (Latency, IOPS) from the system and did robust regression between latency and OIO to come up with a prediction for single workload scenario.

4.2 Dependency of Maximum IOPS on Workload Characteristics

Latency vs OIO relationship as described in Section 4.1 depends on a number of system and workload factors. However, our study showed that workload characteristics dictate this line between Latency vs OIO when system configuration does not change. Our experience with customer systems shows that the majority of workloads show variation of intensity (over a week or during holidays and peak hours etc) but there is generally no change in characteristics (read/write sizes, read/write ratio, sequential/random ratio), we did come across some real life workloads (Exchange, Financial) that exhibit large variations in the characteristics causing wide and divergent Latency vs OIO behavior as seen in Figure 4.2.

As is evident from Figure 4.2, observations in different IO size buckets are clearly segregated in separate regions. For such cases, we give different estimates based on currently observed workload characteristics. We bin workload parameters in different buckets and we estimate the maximum possible IOPS for each bucket separately. When a workload has several such buckets of characteristics, we advise a set of maximum possible IOPS that are governed by each bucket of workload characteristics.

4.3 Provisioning New Workloads

We show in Section 5.3 that a few workload characteristics (read/write sizes, read%, rand%) were enough to capture the essence of the workload and these characteristics would effectively dictate system performance in terms of maximum possible IOPS. This led us to believe that when workloads are abstracted as a set of characteristics, we could relate estimates across different workloads within an error margin of 15%. We used internal micro-benchmark SIO (Simple Input/Output) for creating tables across these finite characteristics dimensions. We then compared observed maximum when system was

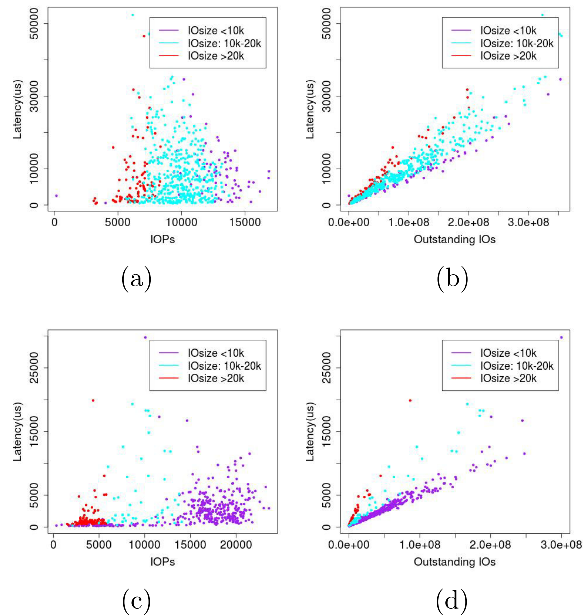


Figure 2: Dependency of the Latency with IOPS and outstanding IO: a) Latency vs IOPS for Exchange workload (b) Latency vs OIO for Exchange workload (c) Latency vs IOPS for Financial workload (d) Latency vs OIO for Financial workload.

driven to saturation with the predicted maximum, and validated the fact that the maximum IOPS is dependent on the workload characteristics for a given system.

We thus extended the workload characteristics table approach to be able to provide maximum possible IOPS estimates before provisioning a new workload. One of the major drawbacks of approach in Section 4.1 is that we need periodic measurements of certain metrics from the system. In other words, we need the workload to be deployed before an estimate can be given. However, because workload characteristics could abstract the workload as seen by the system (Section 4.2), we could give an estimate prior to provisioning the workload if the workload characteristics (read/write sizes, read%, random% etc.) are known.

4.4 Interference of Multiple Workloads

For mixed workload modeling, we view the storage system channel as a black-box serving IOs consisting of the controller, disks, CPU, memory, cache and other architectural components.

We model the black-box as a single server multi-queue where different queues represent different workloads running on different volumes. Service rates for different queues are different depending on the respective workload characteristics.

Let λ be arrival rate of requests and μ be the service rate of the system for a certain workload. Effectively, λ is current IOPS being served for that workload and μ is the maximum possible IOPS if the workload is run on the system in stand-alone mode. In other words, we can consider that no more than μ IOPS of this workload can be pushed in the black-box because it is being utilized 100% by this workload. This is the same as parameter $1/a$ in Section 4.1. Current utilization (ρ) of the black box for this workload is given as

$$Utlization = \rho = \frac{\lambda}{\mu} \qquad (5)$$

In Equation (5), $\lambda = current\ IOPS$ and $\mu = Maximum\ IOPS$.

Given a system with n different workloads provisioned, total utilization is

$$\rho = \sum_{i=1}^{n} \rho_i \quad \text{where} \quad \rho_i = \frac{\lambda_i}{\mu_i} \qquad (6)$$

Applying this for new workload provisioning, if the current utilization for the system (ρ) and service rate for new workload (μ_{new})are known, we have

$$Maximum\ IOPS = (1 - \rho) \times \mu_{new} \qquad (7)$$

Maximum IOPS possible for a workload then essentially depends on how much the storage system is already utilized. For example, if current utilization is 50% then we can have only 50% of the maximum IOPS that were possible in stand-alone mode. Equation(7) is agnostic of the workload type that is running on the storage system. It only requires service rates for different workloads, and service rates depend on workload characteristics. We, therefore, compute service rates for different workload types and then compute resultant utilization of the storage system. Once the total utilization is known, the residual utilization governs the maximum possible IOPS for a new workload depending on the respective service rate.

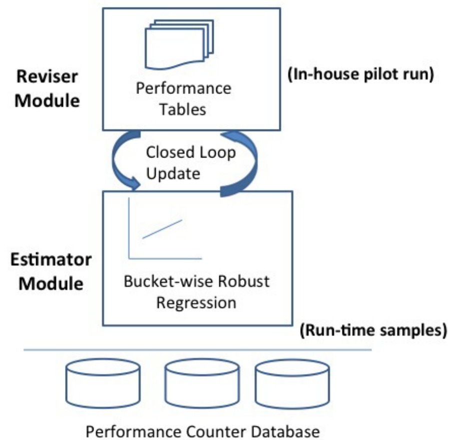


Figure 3: A schematic view of the deployment architecture for dynamic provisioning system

4.5 Deployment and Workflow

The deployment architecture takes care of adaptively adjusting service rate estimates by incorporating feedback from environment. It has two modules as shown in Figure 3. One module (Reviser) stores the workload characteristics based service rate tables as the master tables and a set of active tables. Active tables are derived from running workloads by mining them in workload parameters. At any point of time, this module derives the utilization of the storage system viewing it as a black-box and then predicts the maximum possible IOPS for a workload. The other module (Estimator) stores performance metrics measured from the system. Active tables are dynamically updated to take care of factors like aging, configuration change, system upgrades etc.

When a system is commissioned, master performance-tables are created (in-house pilot run) and populated in the Reviser module.

When workloads start running, performance counters are captured and recent snapshots of performance counters are populated in Estimator module. In Estimator module, workloads are bucketed in various buckets according to workload characteristics. Each bucket has several samples of performance counter measurements. (Workloads may not consist of buckets of several characteristics and therefore several buck-

ets may be empty.) Once performance counters are collected over a period of time, Latency vs OIO curves are estimated for non-empty buckets in Estimator module. From these curves, the maximum IOPS is estimated for the non-empty buckets for the running workload. Estimator module sends values of estimated maximum IOPS for non-empty buckets to Reviser module. Reviser module then compares estimates with those in active tables and incrementally modifies active tables. In summary, dynamic estimates for new or existing workloads are provided based on active tables and current black-box utilization along with knowledge of workload characteristics.

5 Experimental Results

We experimented and validated the effectiveness of proposed provisioning solution for various workloads on two different enterprise storage clusters. We used linux client to send IO traffic to storage servers.

5.1 Workloads

We used a well-cited collection of storage traces released by Microsoft Research (MSR) in Cambridge [18, 20] in 2007 for most of our evaluation. MSR traces record disk activity (captured beneath the file-system cache) of 13 servers with a combined total of 36 volumes for a week. We worked with 4 MSR workloads, namely, TPCE, Exchange, Financial, and a web server (Web). TPCE [3]is a On-Line Transaction Processing(OLTP) Workload developed by Transaction Processing Performance Council(TPC). Exchange workload captures activity of Microsoft Exchange application. Web workload records activity of web servers. Financial transactions are recoded in Financial workload. Raw traces comprise of 10-50 million records and consume just over 150 − 500 MB in compressed CSV format each. Figure 4 shows extracted workload characteristics (read/write sizes, read/random percentage) for these workloads. We replay these traces using a trace replayer that runs on a host. It takes disk number, byte offset, IO size, elapsed time, timestamp and the type of operation (read or write) as input parameters and generates respective workloads.

Apart from these real life workloads, we also used an internal micro-benchmark (derived from

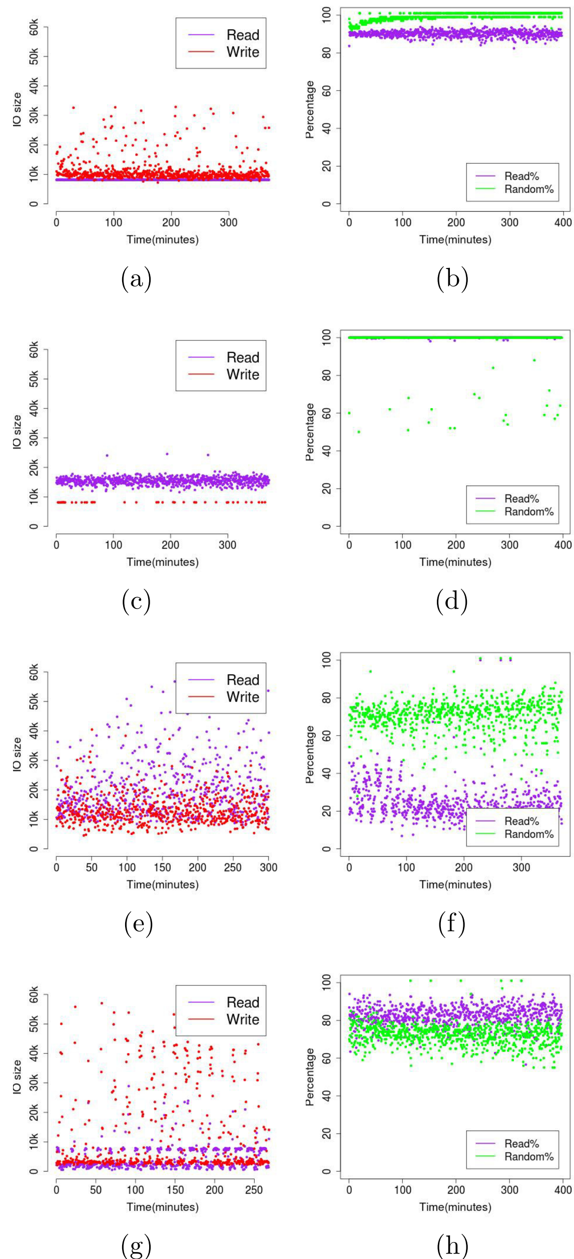


Figure 4: Workload characteristics for different workloads: (a) TPCE IO size (b) TPCE read % and random % (c) Web IO size (d) Web read % and random % (e) Exchange IO size (f) Exchange read % and random % (g) Financial IO size (h) Financial read % and random %

FIO) named SIO [4] (Simple Input/Output). SIO, a synthetic tool available from host, takes read%, random%, IO size, offsets as input custom tun-

[3]URL:http://www.tpc.org/tpce/

[4]URL: http://web.stanford.edu/group/storage /netapp/sio_ntap/siontap.htm

able parameters to send traffic with desired workload characteristics.

As mentioned before, we aimed for an error margin of less than 20%. For all cases, we saturated the system with workload(s) in question and observed the maximum. Prediction error was calculated as below:

$$Prediction\ Error\% = \frac{OM - PM}{OM} \times 100 \quad (8)$$

where OM = Observed Maximum and PM = Predicted Maximum.

5.2 Maximum IOPS for a Single Workload

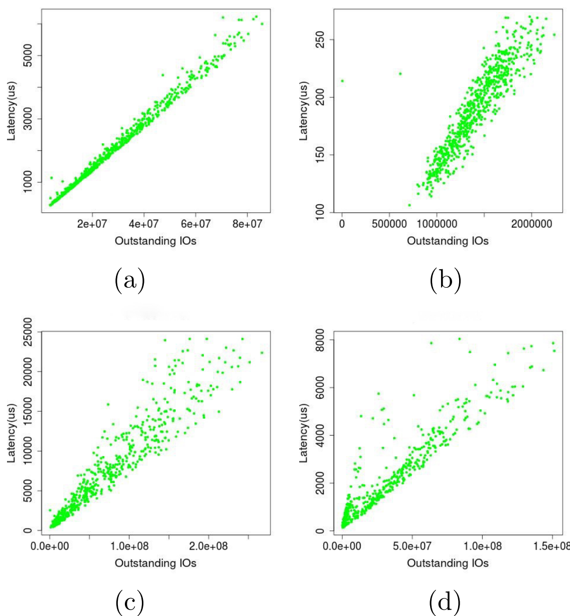


Figure 5: Latency vs OIO (a) TPCE (b)Web (c) Exchange (d)Financial

As seen from workload characteristics in Figure 4, workload parameters like read%, random% and read/write size do not show much variation for TPCE and Web workloads. For these well-behaving workloads, we see linear Latency vs OIO behavior as expected (Figure 5). Error between predicted and observed maximum reduced as we considered more and more sample points to finally stabilize at less than 15%, as seen in Figure 6.

For Exchange and Financial workloads, workload parameters show large variation (Figure 4) that causes scattered and wide-spread Latency vs OIO behavior as seen from the Figure 5. However, recalling from Section 4.2 when these characteristics were further bucketed in various workloads bins and our technique was applied on a region by region basis, we found that the error between the observed and predicted maximum IOPS was less than 15%.

Dependency of maximum IOPS to workload characteristics led us into exploring this modeling further using SIO custom workload generator. We observed that the maximum IOPS on a given storage system is agnostic of the workload type (e.g., Exchange, Web, or Financial) and highly depends on the workload characteristics for a given system. We used an internal micro-benchmark SIO to generate combinations of workload parameters and saturate the system in each case. We collected the performance counters to measure the workload characteristics for each case to do robust regression as per the technique. Table 1 shows the observed maximum values for all buckets. We divided read % range in 6 buckets and IO size range in 5 buckets as shown in Table 1. Table 2 shows predicted maximum (from Latency-OIO regression) values. As seen from Table 3, when workloads are characterized in bins, errors are within acceptable limits. (We have presented the worst error case in tables 1, 2, 3 i.e. SIO random%=0.) This also proved that a workload-aware approach works well for performance prediction.

5.3 New Workload Provisioning

In this section, we extend the results to give predictions for workloads without having to deploy them. This is illustrated with Figure 6.

Workload characteristics for TPCE are read %=80 , random%=100 and IO size = 8k, as seen from Figure 4, If that particular bin from Table 4 is looked up, we get an estimate of 14877 , which is within 5% error margin of what we observed for TPCE when the system was saturated.

We performed with 10% error margin for web workload as well. As the exact web characteristics (Read%=90, Random%=100, IO size=16k) are not available in the table, we average estimates for two bins. So, the maximum IOPS estimate from Table 4 is $(8603 + 7123)/2 = 7863$. This predicted maximum is within 5% error margin to the observed maximum when the workload is actually provisioned (Figure 6(b)).

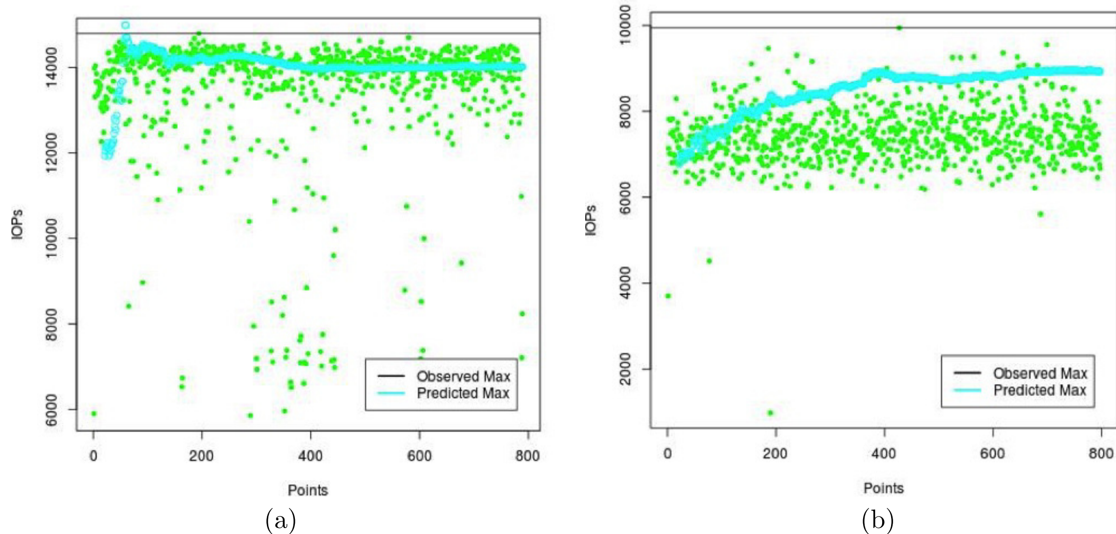


Figure 6: Observed and predicted maximum IOPS for (a)TPCE and (b)Web.

Read % / IO size	0%	20%	40%	60%	80%	100%
4k	20656	20096	20325	20658	20679	21367
8k	13931	15354	15341	15045	14684	12698
16k	7155	8794	10608	9833	8476	7123
32k	3593	4479	5804	5715	4398	3583
64k	1802	2258	2867	2820	2228	1804

Table 1: Observed maximum IOPS for different buckets of workload characteristics.

Read % /IO size	0%	20%	40%	60%	80%	100%
4k	16575	16430	14301	17625	17997	18688
8k	12405	13016	11714	13006	14179	11752
16k	7130	8456	9684	9847	8121	6985
32k	3620	4422	5445	5684	4400	3605
64k	1783	2229	2760	2851	2183	1758

Table 2: Predicted maximum IOPS for different buckets of workload characteristics.

Read % / IO size	0%	20%	40%	60%	80%	100%
4k	13.07	18.46	28.12	11.76	13.58	12.97
8k	11.41	15.23	21.77	12.53	4.70	7.94
16k	0.24	4.61	8.64	1.56	5.60	1.93
32k	-0.42	0.85	6.96	1.60	2.46	-0.28
64k	1.17	1.15	4.37	-0.64	3.99	2.74

Table 3: Error % in prediction of the maximum IOPS for different workload characteristics.

Read % / IO size	0%	20%	40%	60%	80%	100%
4k	19068	20150	19896	19975	20825	21473
8k	14002	15355	14974	14869	14877	12765
16k	7147	8864	10600	10003	8603	7123
32k	3605	4460	5852	5776	4511	3595
64k	1805	2255	2886	2833	2274	1808

Table 4: New Workload Provisioning: Master Table (SIO Random% = 100)

Run Details	TPCE1	TPCE2	Web1	Web2	SIO	Utilization
Service Rate	14k	14k	8k	8k	8k	1
F=TPCE, B=Web	4k	4k	1.2k	1.2k	1.2k	1.02
F=Web, B=TPCE	1k	1k	3k	3k	1k	1.03
F=TPCE+Web	2.3k	2.3k	2.3k	2.3k	0.6k	0.98

Table 5: Multiple Workloads Scenarios(F= Foreground, B=Background)

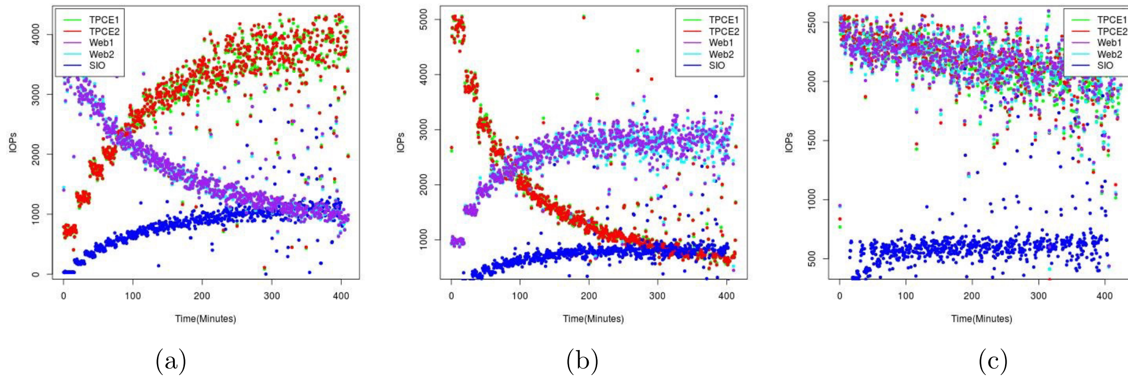


Figure 7: Multiple Workloads Scenarios: (a) Foreground=TPCE, Background=Web (b) Foreground=Web,Background=TPCE (c)Foreground=TPCE+Web

Volume	WL1	WL2	WL3	WL4	WL5
Service Rate	19975	14869	10003	5776	2833
Current IOPS	3200	1700	800	200	1730 (estimated maximum)

Table 6: Estimated maximum IOPS for a new workload (WL5) in presence of four other workloads.

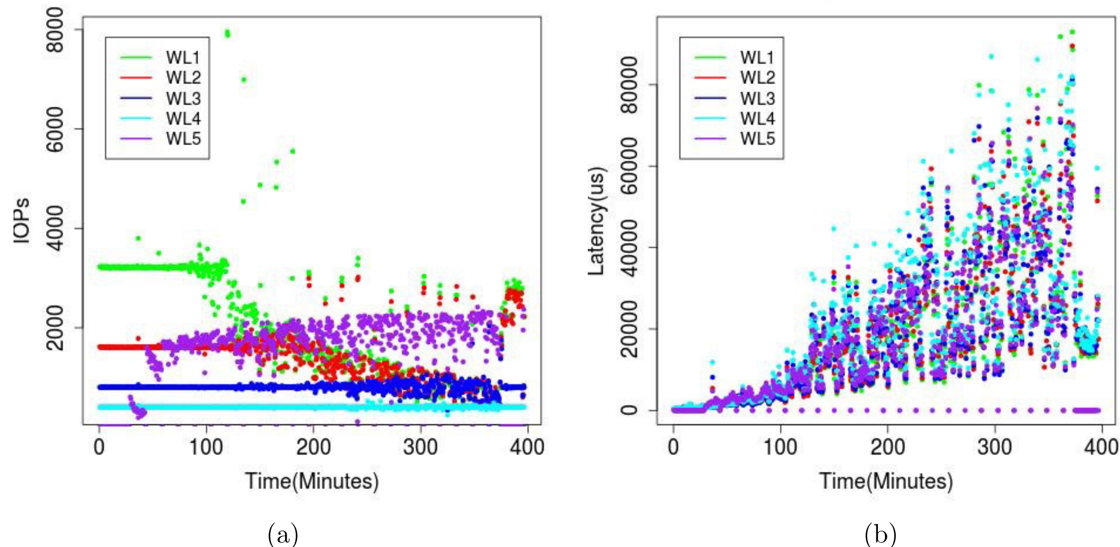


Figure 8: Effect of a new workload(WL5) on existing workloads

5.4 Interference of Multiple Workloads

For studying the interference of multiple workloads based on black-box utilization model, we created five volumes on the same storage aggregate and used pre-determined (using earlier techniques) service rates in order to calculate to-

tal system utilization for each of the three scenarios as listed in Table 5. F denotes workloads in the foreground which were increased in intensity continuously and B denotes workloads in the background that were kept constant in intensity.

In all scenarios, we saturated the system with foreground workloads. Figure 7 shows how IOPS changed for all workloads. In Table 5, we have IOPS in saturated condition for each of them. The last column denotes total system utilization calculated according to Equation (6). As expected, the total utilization number hovers around 1 because the system was saturated with foreground workloads.

We then created real life scenario [5] by deploying four volumes with four different SIO workloads WL1-WL4. On these volumes, we sent traffic of constant intensity (IOPS were kept constant). On fifth volume, we started increasing intensity of WL5. Figure 8 shows how IOPS and latency changed for all 5 workloads. From Table 6, service rate row contains values for service rate as obtained from SIO micro-benchmark table. Our aim is to calculate the maximum number of IOPS possible for WL5 given the current mix of other workloads WL1-WL4.

According to Equation (6), we can obtain the current utilization of the black-box using Table 6. So, current utilization = 3200/19975 + 1700/14869 + 800/10003 + 200/5776 = 0.39. Therefore, the predicted maximum IOPS according to Equation (7) for WL5 is = (1- 0.39) * 2833 = 1730. This is evident from Figure 8 (a). As IOPS for WL5 (purple) increased beyond the estimated maximum IOPS 1730 (this is approximately at point around 120 minutes on X-axis), IOPS for other workloads WL1-WL4 started decreasing gradually (10% in 10 minutes) and latencies abruptly shot up (50% in 10 minutes) after 120 minutes mark (Figure 8 (b))

5.5 Feedback Mechanism

File system fragmentation is known to affect the performance of a storage system [6]. We periodically fragmented the file-system in eight rounds using a synthetic tool. We set fragmentation parameters to most severe levels to simulate rapid aging of file-system. Each workload was increased in intensity during each round. We observed number of IOPS decreasing due to the effect of fragmentation. We periodically collected counters and predicted maximum IOPS for each workload using Equation (4). We then used a simple lazy update heuristic to update our estimates considering predicted maximum IOPS during consecutive iterations. Table 7 shows the baseline estimates before file-system aging and observed maximum at the end of eight rounds. There is a large difference between these numbers because of fragmentation. However, closed loop update mechanism updated estimate is a lot closer (around 10% error) to observed end maximum.

[5] As observed from customer systems

[6] URL:https://en.wikipedia.org/wiki/File_system_fragmentation

6 Summary and Future Work

In this paper, we present a mechanism for combining a queuing model with machine learning for dynamic provisioning of storage workloads. We estimate the maximum possible IOPS for a running workload using robust regression viewing the storage system as a black-box. For new workloads, we devise a method based on study of workload characteristics . We account for interference of existing workloads using the utilization of the storage server viewing the entire system as a multi-queue-single-server model where the queues are independent of each other. We have also developed a feedback mechanism to adapt estimates for change in factors like configuration change and aging etc. In all above cases, we were able to provide estimates within a reasonable error margin of 15-20%.

The techniques developed as part of this work are extensible to provide more comprehensive solutions for storage provisioning.

- Optimal Provisioning: Standard optimization techniques in literature can be used to formulate optimization objectives around performance and utilization. That would mean we could come up with the best possible arrangement and redistribution recommendations for workloads in or across storage clusters.
- Service Rate Normalization: For mixed workload modeling, we have used already estimated service rates. These can also be looked up in service rate tables if the workload characteristics are known. If the workload characteristics are unknown, service rates can be normalized dynamically to be able to apply the technique for migration.

Volume	Baseline	Updated End Prediction	Observed End Maximum	Error %
WL1	3244	2678	2433	10.07
WL2	2992	1496	1340	11.66
WL3	2793	790	718	10.04
WL4	2373	402	424	4.96
WL5	1766	209	216	2.90

Table 7: Feedback update for file-system fragmentation

Acknowledgements

The authors would like to acknowledge Veena Bhat, an intern with Advanced Technology Group, who helped us with scripts and execution part. Special thanks are due to Ajay Bakhshi and Siddhartha Nandi for facilitating this project and coming up with insightful suggestions to take this work forward in meaningful directions. The authors are very thankful to their shephard Chad Verbowski for critically evaluating the paper.

References

[1] Basak, J., Wadhwani, K., Voruganti, K., Narayanamurthy, S., Mathur, V., and Nandi, S. Model building for dynamic multi-tenant provider environments. *ACM SIGOPS Operating Systems Review 46* (2012), 20–31.

[2] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[3] Breiman, L. Bagging predictors. *Machine learning 24* (1996), 123–140.

[4] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. Chapman & Hall, New York, 1983.

[5] Coffman, E. G. Queueing models of secondary storage devices. Tech. Rep. 90-943, Department of Computer Science, Purdue University, USA, 1990.

[6] Gulati, A., Ahmad, I., and Waldspurger, C. Parda: Proportional allocation of resources for distributed storage access. In *FAST09, San Jose, CA* (2009), pp. 85–98.

[7] Gulati, A., Kumar, C., Ahmad, I., and Kumar, K. Basil: Automated io load balancing across storage devices. In *Proc. 8th USENIX Conf File and Storage Technologies (FAST)* (2010), pp. 169–182.

[8] Gulati, A., Merchant, A., and Varman, P. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI 10, Vancouver, Canada* (2010).

[9] Gulati, A., Shanmuganathan, G., Ahmad, I., Waldspurger, C. A., and Uysal, M. Pesto: Online storage performance management in virtualized datacenters. In *Proc. 2nd ACM Symp. Cloud Computing (SOCC '11)* (2011), pp. 19–32.

[10] Johnson, T. Queuing models of tertiary storage. In *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies* (1996), vol. NASA-CP-3340-Vol- 2, pp. 529–552.

[11] Merchant, A., Uysal, M., Padala, P., Zhu, X., Singhal, S., and Shin, K. Maestro: Quality-of-service in large disk arrays. In *ICAC 11, Karlsruhe, Germany* (2011), pp. 245–254.

[12] Mesnier, M., Wachs, M., Sambasivan, R. R., Zheng, A. X., and Ganger, G. Modeling the relative fitness of storage. In *Proc. Int Conf Measurements and Modeling of Computer Systems, SIGMETRICS 2004* (2004), pp. 37–48.

[13] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. Adaptive control of virtualized resources in utility computing environments. In *EuroSys 07, Lisbon, Portugal* (2007), pp. 289–302.

[14] Pentakalos, O. I., Menasce, D. A., Halem, M., and Yesha, Y. Analytical performance modeling of hierarchical mass storage systems. *IEEE Trans. Computers 46* (1997), 1103–1118.

[15] Rasmussen, C. E., and Williams, C. K. I. *Gaussian Processes for Machine Learning*. MIT Press, USA, 2006.

[16] Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P. Design and analysis of computer experiments. *Statistical Science 4* (1989), 409435.

[17] Smola, A. J., and Schölkopf, B. A tutorial on support vector regression. Tech. Rep. NC-TR-98-030, NeuroCOLT, Royal Holloway College, University of London, UK, 1998.

[18] SNIA. Snia iotta repository, 2011. http://iotta.snia.org/trace.

[19] Soundararajan, G., and Amza, C. Towards end-to-end quality of service: Controlling i/o interference in shared storage servers. In *Middleware* (2008), pp. 287–305.

[20] SPC. Storage Performance Council: SPC trace file format specification, 2002. http://skuld.cs.umass.edu/traces/storage/SPC-Traces.pdf.

[21] Trushkowsky, B., Bod?k, P., Fox, A., Franklin, M., Jordan, M., and Patterson, D. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST'11* (2011), pp. 163–176.

[22] Vapnik, V., Golowich, S., and Smola, A. Support vector method for function approximation, regression estimation, and signal processing. In *Advances in Neural Information Processing Systems 9*, M. Mozer, M. Jordan, and T. Petsche, Eds. MIT Press, Cambridge, MA, USA, 1997, pp. 281–287.

[23] Wang, A., Venkataraman, S., Alspaugh, S., Katz, R., and Stoica, I. Enabling high-level slos on shared storage systems. In *Symposium on Cloud Computing* (2012).

[24] Wang, A., Venkataraman, S., Alspaugh, S., Stoica, I., and Katz, R. Sweet storage slos with frosting. In *HotCloud'12, Boston, MA* (2012).

[25] Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G. R. Storage device performance prediction with cart models. In *Proc. Int Conf Measurements and Modeling of Computer Systems, SIGMETRICS 2004* (2004), pp. 412–413.

[26] Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G. R. Storage device performance prediction with cart models. Tech. Rep. CMU-PDL-04-103, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, USA, 2004.

[27] Williams, C. K. I. Prediction with gaussian processes: From linear regression to linear prediction and beyond. In *Learning and Inference in Graphical Models* (1998), Kluwer, pp. 599–621.

[28] Yin, L., Uttamchandani, S., and Katz, R. An empirical exploration of black-box performance models for storage systems. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)* (2006), pp. 433 – 440.

[29] Zhang, L., Liu, G., Zhang, X., Jiang, S., and Chen, E. Storage device performance prediction with selective bagging classification and regression tree. *Lecture Notes in Computer Science: Proceedings of the 2010 IFIP international conference on Network and parallel computing 6289* (2010), 121–133.

SF-TAP: Scalable and Flexible Traffic Analysis Platform running on Commodity Hardware

Yuuki Takano[1,2]
ytakano@wide.ad.jp

Ryosuke Miura[1,2]
myu2@nict.go.jp

Shingo Yasuda[1,2]
s-yasuda@nict.go.jp

Kunio Akashi[2]
k_akashi@jaist.ac.jp

Tomoya Inoue[2,1]
t-inoue@jaist.ac.jp

National Institute of Information and Communications Technology, Japan[1]

Japan Advanced Institute of Science and Technology[2]

Abstract

Application-level network traffic analysis and sophisticated analysis techniques such as machine learning and stream data processing for network traffic require considerable computational resources. In addition, developing an application protocol analyzer is a tedious and time-consuming task. Therefore, we propose a scalable and flexible traffic analysis platform (SF-TAP) that provides an efficient and flexible application-level stream analysis of high-bandwidth network traffic. Our platform's flexibility and modularity allow developers to easily implement multicore scalable application-level stream analyzers. Furthermore, SF-TAP is horizontally scalable and can therefore manage high-bandwidth network traffic. We achieve this scalability by separating network traffic based on traffic flows, forwarding the separated flows to multiple SF-TAP cells, each of which consists of a traffic capturer and application-level analyzers. In this study, we discuss the design and implementation of SF-TAP and provide details of its evaluation.

1 Introduction

Network traffic engineering, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), and the like perform application-level network traffic analysis; however, this analysis is generally complicated, requiring considerable computational resources. Therefore, in this paper, we propose a scalable and flexible traffic analysis platform (SF-TAP) that runs on commodity hardware. SF-TAP is an application-level traffic analysis platform for IDSs, IPSs, traffic engineering, traffic visualization, network forensics, and so on.

Overall, two problems arise in application-level network traffic analysis. The first is the difficulty in managing various protocols. There are numerous application protocols, with new protocols being defined and implemented every year. However, the implementation of application protocol parsers and analyzing such programs is a tedious and time-consuming task. Thus, a straightforward implementation of a parser and analyzer is crucial for application-level network traffic analysis. To enable such a straightforward implementation, approaches using domain-specific languages (DSLs) have been previously proposed. For example, BinPAC [24] is a parser for application protocols that are built into Bro IDS software. Wireshark [38] and Suricata [34] are binding Lua languages, with analyzers that can be implemented in Lua. Unfortunately, DSLs are typically not sufficiently flexible because there is often the requirement that researchers and developers want to use specific programming languages for specific purposes such as machine learning.

The second problem with application-level network traffic analysis is the low scalability of conventional software. Traditional network traffic analysis applications such as tcpdump [35], Wireshark, and Snort [33] are single threaded and therefore cannot take advantage of multiple CPU cores when performing traffic analysis. With the objective of improving the utilization of CPU cores, several studies have been conducted and software solutions have been proposed. For example, for high-bandwidth and flow-based traffic analysis, GASPP [36] exploits GPUs and SCAP [25], which utilizes multiple CPU cores, implements a Linux kernel module. Although it is important to reconstruct TCP flows efficiently, the efficiency of a parser or analyzing programs is more critical because they require more computational resources for performing such deep analysis as pattern matching or machine learning. Therefore, multicore scaling is required for both TCP flow reconstruction and traffic-analyzing components to enable the analysis of high-bandwidth traffic. In addition to multicore scalability, horizontal scalability is important for the same reason. To support the deep analysis of high-bandwidth network traffic to be performed easily and cost effectively, the corresponding application-level analysis plat-

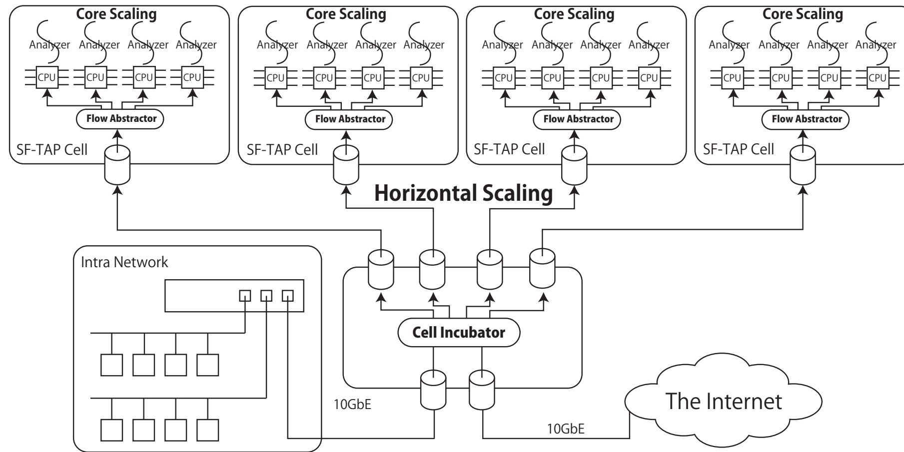


Figure 1: High-level Architecture of SF-TAP

form must have horizontal scalability.

Given the abovementioned issues, in this study, we discuss the design and implementation of SF-TAP for high-bandwidth application-level traffic analysis. SF-TAP adopts a flow abstraction mechanism that abstracts network flows by files, much like Plan 9 [28], UNIX's /dev, or the BSD packet filter (BPF) [19]. Using the interfaces, analyzing logic developers can rapidly and flexibly implement application-level analyzers in any language. Furthermore, L3/L4-level controlling and application-level analyzing components are separate given the modularity of the architecture. As a result of this design, analyzing components can be flexibly implemented, dynamically updated, and multicore scalable.

Note that our proof-of-concept implementation is distributed on the Web (see [32]) under BSD licensing for scientific reproducibility, and thus, it is freely available for use and modification.

2 Design Principles

In this section, we discuss the design principles of SF-TAP, i.e., the abstraction of network flows, multicore scalability, horizontal scalability, and modularity.

First, we describe the high-level architecture of SF-TAP, which is shown in Figure 1. SF-TAP has two main components: the cell incubator and flow abstractor. We call a group that consists of a flow abstractor and analyzers a cell. The cell incubator provides horizontal scalability; thus, it captures network traffic, separates it on the basis of the flows, and forwards separated flows to specific target cells. Conventional approaches using pcap or other methods cannot manage high-bandwidth network traffic, but our approach has successfully managed 10 Gbps network traffic using netmap [30], multiple threads, and lightweight locks.

Furthermore, by separating network traffic, we can manage and analyze high-bandwidth network traffic using multiple computers. By providing multicore scalability, the flow abstractor receives flows from the cell incubator, reconstructs TCP flows, and forwards the flows to multiple application-level analyzers. The multicore and horizontally scalable architectures enable application-level traffic analysis, which requires considerable computational resources, to be performed efficiently.

2.1 Flow Abstraction

DSL-based approaches have been adopted in several existing applications, including the aforementioned Wireshark, Bro, and Suricata. However, these approaches are not always appropriate because different programming languages are suitable for different requirements. As an example, programming languages suitable for string manipulation, such as Perl and Python, should be used for text-based protocols. Conversely, programming languages suitable for binary manipulation, such as C and C++, should be used for binary-based protocols. Furthermore, programming languages equipped with machine learning libraries should be used for machine learning.

Therefore, we propose an approach that abstracts network flows into files using abstraction interfaces, much like Plan 9; UNIX's /dev; and BPF, to provide a flexible method for analyzing application-level network traffic. Using these abstraction interfaces, various analysts such as IDS/IPS developers or traffic engineers can implement analyzers using their preferred languages. Flexibility is of particular importance in the research and development phase of traffic analysis technologies.

2.2 Multicore Scalability

To analyze high-bandwidth network traffic efficiently, many CPU cores should be utilized for the operation of both TCP/IP handlers and analyzers. We achieve multicore scalability through our modular architecture and threads. More specifically, the flow abstractor is multithreaded with modularity that allows analyzers to be CPU core scalable.

2.3 Horizontal Scalability

Application-level analyzers require substantial computational resources. For example, string parsing is used to analyze HTTP messages, pattern matching via regular expressions is used to filter URLs in real time, and machine learning techniques are applied to extract specific features of network traffic. In general, these processes consume a considerable amount of CPU time.

Accordingly, we propose a horizontally scalable architecture for high-bandwidth application-level traffic analysis. The horizontal scalability allows the analyzers, which consume considerable computational resources, to be operated on multiple computers.

2.4 Modular Architecture

The modularity of our architecture is an important factor in providing multicore scalability and flexibility. In addition, it offers some advantages. In the research and development phase, analyzing components are frequently updated. However, if an update is required, traditional network traffic analysis applications, which are monolithic, such as Snort or Bro, must halt operation of all components, including the traffic capturer.

We therefore propose a modular architecture for network traffic analysis that allows network traffic capturing and analyzing components to be separate. Thus, modularity allows traffic analysis components to be easily updated without impacting other components. Furthermore, bugs in applications still under development do not negatively affect other applications.

2.5 Commodity Hardware

Martins et al. [18] indicated that hardware appliances are relatively inflexible and the addition of new functions is not easily accomplished. Furthermore, hardware appliances are very expensive and not easily scaled horizontally. To address these problems, software-based alternatives running on commodity hardware, including network function virtualization (NFV) [22], are now being developed. We propose a software-based approach that runs in commodity hardware environments to achieve flexibility and scalability similar to those of NFV.

3 Design

In this section, we describe the design of SF-TAP.

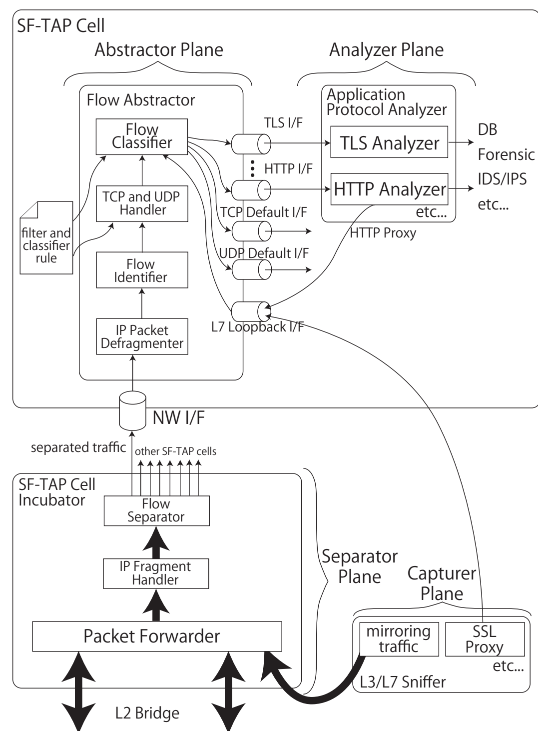


Figure 2: Architecture of SF-TAP

3.1 Design Overview

Figure 2 shows the architecture of SF-TAP. SF-TAP consists of four planes: the capturer, separator, abstractor, and analyzer planes. Each is described below. The capturer plane is a plane for capturing network traffic. More specifically, this plane consists of a port-mirroring mechanism of L2/L3 switches, an SSL proxy to sniff plain text, and so on. The separator plane is a plane that provides horizontal scalability for high-bandwidth network traffic analysis. This plane separates network traffic into L3/L4 levels, forwarding flows to multiple cells, each of which consists of the abstractor and analyzer planes.

The abstractor plane is a plane for network flow abstraction. This plane defragments IP fragmentations, identifies flows at the L3/L4 level, reconstructs TCP streams, detects the application protocol using regular expressions, and outputs flows to the appropriate abstraction interfaces. Traffic analyzer developers can develop analyzers by accessing the interfaces provided by this plane. Finally, the analyzer plane is a plane for analyzers developed by SF-TAP users. Users can implement analyzers in any programming language.

In this study, we focus on the separator and abstractor planes because the components of the capturer plane are well known and analyzers of the analyzer plane are developed by SF-TAP users. In the subsections that follow,

```
1  http:
2    up:      '^[-a-zA-Z]+ .+ HTTP/1\.(0\r?\n|1\r?\n([-a-zA-
   Z]+: .+\r?\n)+)'
3    down:    '^HTTP/1\.[01] [1-9][0-9]{2} .+\r?\n'
4    proto:   TCP  # TCP or UDP
5    if:      http # path to UNIX domain socket
6    nice:    100  # priority
7    balance: 4    # balanced by 4 IFs
8
9  torrent_tracker: # BitTorrent Tracker
10   up:      '^GET .*(announce|scrape).*\?.*info_hash=.+&.+
    HTTP/1\.(0\r?\n|1\r?\n([-a-zA-Z]+: .+\r?\n)+)'
11   down:    '^HTTP/1\.[01] [1-9][0-9]{2} .+\r?\n'
12   proto:   TCP
13   if:      torrent_tracker
14   nice:    90  # priority
15
16 dns_udp:
17   proto:   UDP
18   if:      dns
19   port:    53
20   nice:    200
```

Figure 3: Configuration Example for Flow Abstractor

we describe the design of the flow abstractor and cell incubator and show an example of an analyzer for HTTP.

3.2 Flow Abstractor Design

In this section, we describe the design of the flow abstractor, as shown in Figure 2, and explain its key mechanisms using the example configuration file shown in Figure 3. For human readability, the flow abstractor adopts YAML [39] to describe its configuration. Using a top–down approach, the flow abstractor consists of four components: the IP packet defragmenter; flow identifier; TCP and UDP handler; and flow classifier.

3.2.1 Flow Reconstruction

The flow abstractor defragments fragmented IP packets and reconstructs TCP streams. Thus, analyzer developers need not implement the complicated reconstruction logic required for application-level analysis.

The IP packet defragmenter shown in Figure 2 is a component that performs IP packet defragmentation. Defragmented IP packets are forwarded to the flow identifier, which identifies the flow as being at the L3/L4 level. We identify flows using 5-tuples consisting of the source and destination IP addresses, source and destination port numbers, and a hop count, which is described in Section 3.2.2. After the flows have been identified, the TCP streams are reconstructed by the TCP and UDP handler, and then, the flows are forwarded to the flow classifier.

3.2.2 Flow Abstraction Interface

The flow abstractor provides interfaces that abstract flows at the application level. For example, Figure 2 shows the TLS and HTTP interfaces. Furthermore, in Figure 3, the HTTP, BitTorrent tracker [2], and DNS interfaces are defined.

```
1  $ ls -R /tmp/sf-tap
2  loopback7=       tcp/             udp/
3
4  /tmp/sf-tap/tcp:
5  default=         http2=           ssh=
6  dns=             http3=           ssl=
7  ftp=             http_proxy=      torrent_tracker=
8  http0=           irc=             websocket=
9  http1=           smtp=
10
11 /tmp/sf-tap/udp:
12 default=         dns=             torrent_dht=
```

Figure 4: Directory Structure of Flow Abstraction Interface

The flow classifier classifies flows of various application protocols, forwarding them to flow abstraction interfaces, which are implemented using a UNIX domain socket, as shown in Figure 4. The file names of the interfaces are defined by items of *if*; for example, on lines 5, 13, and 18 in Figure 3, the interfaces of HTTP, BitTorrent tracker, and DNS are defined as `http`, `torrent_tracker`, and `dns`, respectively. By providing independent interfaces for each application protocol, any programming language can be used to implement analyzers.

Further, we designed a special interface for flow injection called the L7 loopback interface, i.e., *L7 Loopback I/F* in Figure 2. This interface is convenient for encapsulated protocols such as HTTP proxy. As an example, HTTP proxy can encapsulate other protocols within HTTP, but the encapsulated traffic should also be analyzed at the application level. In this situation, a further analysis of encapsulated traffic can easily be achieved by re-injecting encapsulated traffic into the flow abstractor via the L7 loopback interface. The flow abstractor manages re-injected traffic in the same manner. Therefore, the implementation of the application-level analysis of encapsulated traffic can be simplified, although, in general, it tends to remain rather complex.

Note that the L7 loopback interface may cause infinite re-injections. To avoid this problem, we introduce a hop count and corresponding hop limitation. The flow abstractor drops injected traffic when its hop count exceeds the hop limitation, thus avoiding infinite re-injection.

In addition to the flow abstraction and L7 loopback interface, the flow abstractor provides default interfaces for unclassified network traffic. Using these default interfaces, unknown or unclassified network traffic can be captured.

3.2.3 TCP Session Abstraction

An example output is shown in Figure 5, with the flow abstractor first outputting a header, which includes information on the flow identifier and abstracted TCP event; the flow abstractor then outputs the body, if it exists.

```
1   ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=CREATED
2   ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=DATA,from=2,match=down,len=1398
3
4   1398[bytes] Binary Data
5
6   ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=DESTROYED
```

Figure 5: Example Output of Flow Abstraction Interface

Line 1 of Figure 5 indicates that a TCP session was established between 192.168.0.1:62918 and 192.168.0.2:80. Line 2 indicates that 1398 bytes of data were sent to 192.168.0.1:62918 from 192.168.0.2:80. The source and destination addresses can be distinguished by the value of *from*, and the data length is denoted by the value of *len*. Lines 3–5 indicate that transmitted binary data are outputted. Finally, line 6 indicates that the TCP session was disconnected.

In the figure, *match* denotes the pattern, i.e., *up* or *down*, shown in Figure 3, that is used for protocol detection.

Managing a TCP session is quite complex; thus, the flow abstractor abstracts TCP states as three events, i.e., *CREATED, DATA, and DESTROYED*, to reduce complexity. Accordingly, analyzer developers can easily manage TCP sessions and keep their efforts focused on application-level analysis.

3.2.4 Application-level Protocol Detection

The flow classifier shown in Figure 2 is an application-level protocol classifier, which detects protocols using regular expressions and a port number. Items *up* and *down*, shown in Figure 3, are regular expressions for application-level protocol detection, and when upstream and downstream flows are matched by these regular expressions, flows are outputted to a specified interface. There are several methods for detecting application-level protocols, including Aho–Corasick, Bayesian filtering, and regular expressions. However, we adopt regular expressions because of its generality and high expressive power. In addition, a port number can be used to classify flows as application-level flows. As an example, line 19 of Figure 3 indicates that DNS is classified by port number 53.

Values of *nice*, which is introduced to remove ambiguity in Figure 3, are used for priority rules; here, the lower the given value, the higher the priority. For example, because BitTorrent tracker adopts HTTP for its communication, there is no difference in terms of protocol formats between HTTP and BitTorrent tracker. Accordingly, ambiguity occurs if rules for HTTP and BitTorrent tracker have the same priority; however, this ambiguity is removed by introducing priorities. In Figure 3, the priority of BitTorrent tracker is configured as being higher than that of HTTP.

3.3 Load Balancing using the Flow Abstraction Interface

In general, the number of occurrences of each application protocol in a network is biased. As such, if only one analyzer process is executed for one application protocol, the computational load will be concentrated in a particular analyzer process. Therefore, we introduce a load-balancing mechanism into the flow abstraction interfaces.

The configuration of the load-balancing mechanism is shown on line 7 of Figure 3. Here, the value of *balance* is specified as 4, indicating that HTTP flows are separated and outputted to four balancing interfaces. Interfaces *http0=, http1=, http2=,* and *http3=* in Figure 4 are the balancing interfaces. By introducing one-to-many interfaces, analyzers that are not multithreaded are easily scalable to CPU cores.

3.4 Cell Incubator Design

The cell incubator shown in Figure 2 is a software-based network traffic balancer that mirrors and separates network traffic based on the flows, thus working as an L2 bridge. The cell incubator consists of a packet forwarder, an IP fragment handler, and a flow separator.

The packet forwarder receives L2 frames and forwards them to the IP fragment handler. Furthermore, it forwards frames to other NICs, such as the L2 bridge, if required. Consequently, SF-TAP can be applied without hardware-based network traffic mirroring.

An IP fragment handler is required for the flow separation of fragmented packets because these packets do not always include an L4 header. This component identifies packets based on the given flows even if the packets are fragmented, forwarding the packets to the flow separator.

The flow separator forwards the packets to multiple SF-TAP cells using flow information that consists of the source and destination IP addresses and port numbers. The destination SF-TAP cell is determined by the hash value of the flow identifier.

3.5 HTTP Analyzer Design

In this subsection, we describe the design of an HTTP analyzer, which is an example of an application-level analyzer. The HTTP analyzer reads flows from the abstraction interface of HTTP provided by the flow abstractor

```
1  {
2   "client": {
3     "port": "61906",
4     "ip":"192.168.11.12",
5     "header": {
6       "host": "www.nsa.gov",
7       "user-agent":"Mozilla\/5.0 (Macintosh; Intel Mac OS
  X 10.9; rv:31.0) Gecko\/20100101 Firefox\/31.0",
8       "connection": "keep-alive",
9       "pragma": "no-cache",
10      "accept": "text\/html,application\/xhtml+xml,
application\/xml;q=0.9,*\/*;q=0.8",
11      "accept-language": "ja,en-us;q=0.7,en;q=0.3", 11 "
accept-encoding": "gzip, deflate",
12      "cache-control": "no-cache"
13    },
14    "method": {
15      "method": "GET",
16      "uri": "\/",
17      "ver": "HTTP\/1.1"
18    },
19    "trailer": {}
20  },
21  "server": {
22    "port": "80",
23    "ip": "23.6.116.226",
24      "header": {
25      "connection": "keep-alive",
26      "content-length":"6268",
27      "date": "Sat, 16 Aug 2014 11:38:25 GMT",
28      "content-encoding": "gzip",
29      "vary": "Accept-Encoding",
30      "x-powered-by": "ASP.NET",
31      "server": "Microsoft-IIS\/7.5",
32      "content-type": "text\/html"
33    },
34    "response": {
35      "ver": "HTTP\/1.1",
36      "code": "200",
37      "msg": "OK"
38    },
39    "trailer": {}
40  }
41 }
```

Figure 6: Example Output of HTTP Analyzer

and then serializes the results into JSON format to provide a standard output. Figure 6 shows an example output of the HTTP analyzer. The HTTP analyzer reads flows and outputs the results as streams.

4 Implementation

In this section, we describe our proof-of-concept implementation of SF-TAP.

4.1 Implementation of the Flow Abstractor

We implemented the flow abstractor in C++; it depends on Boost [3], libpcap [35], libevent [14], RE2 [29], and yamp-cpp [40] and is available on Linux, *BSD, and MacOS X. The flow abstractor is multithreaded, with traffic capture, flow reconstruction, and application protocol detection executed by different threads. For simplicity and clarity, we applied a producer–consumer pattern for data transfer among threads.

The flow abstractor implements a garbage collector for zombie TCP connections. More specifically, TCP connections may disconnect without an FIN or RST packet because of PC or network troubles. The garbage collector collects this garbage on the basis of timers, adopting a partial garbage collection algorithm to avoid locking for a long time period.

In general, synchronization among threads requires much CPU loads. Thus, in the flow abstractor, we implemented bulk data transfers among threads. More specifically, bulk data transfers are performed among threads if the specified amount of data is in the producer's queue or the specified time has elapsed.

The performances of netmap [30] and DPDK [8] are better than libpcap; however, we did not adopt them because of their higher CPU resource consumption and less flexibility. Note that netmap [1] and DPDK require significant CPU resources because they access network devices via polling to increase throughput. Accordingly, they take away CPU resources from application-level analyzers, which require a substantial amount of CPU resources. Furthermore, netmap and DPDK exclusively attach to NICs; thus, other programs such as tcpdump cannot attach to the same NICs. This is an annoyance for network operations and for developing or debugging network software. High throughput, if required, can be accomplished with the help of netmap–libpcap [21].

4.2 Implementation of the Cell Incubator

The cell incubator must be able to manage high-bandwidth network traffic, but conventional methods such as pcap cannot manage high bandwidth. Therefore, we took advantage of netmap for our cell incubator implementation to provide packet capturing and forwarding. Consequently, we could implement a software-based high-performance network traffic balancer, i.e., the cell incubator.

The cell incubator is implemented in C++ and is available on FreeBSD and Linux. It has an inline mode and a mirroring mode. The inline mode is a mode in which the cell incubator works as an L2 bridge. On the other hand, the mirroring mode is a mode in which the cell incubator only receives and separates L2 frames, i.e., it does not bridge among NICs (unlike the L2 bridge). Users can select either the inline or mirroring mode when deploying the cell incubator.

The separation of network traffic is performed using hash values of each flow's source and destination IP addresses and port numbers. Thus, an NIC to which a flow is forwarded is uniquely decided.

Because we adopted netmap, we require that the NICs used by the cell incubator are netmap-available. In general, receive-side scaling (RSS) is enabled on NICs that are netmap-available, and there are multiple receiving

[1] netmap can manage packets by blocking and waiting, but this increases latency.

and sending queues on the NICs. Thus, the cell incubator generates a thread for each queue to balance the CPU load and achieve high-throughput packet managing. However, sending queues are shared among threads; thus, exclusive controls, which typically require a heavy CPU load, are needed. Therefore, to reduce the CPU load for exclusive controls, we adopted a lock mechanism that takes advantage of the compare-and-swap instruction.

4.3 Implementation of the HTTP Analyzer

For demonstration and evaluation, we implemented an HTTP analyzer, comprising only 469 lines, in Python. In our implementation, TCP sessions are managed using Python's dictionary data structure. The HTTP analyzer can also be easily implemented in other lightweight languages. Note that the Python implementation was used for performance evaluations presented in Section 5.

5 Experimental Evaluation

In this section, we discuss our experimental evaluations of SF-TAP.

5.1 HTTP Analyzer and Load Balancing

A key feature of the flow abstractor is its multicore scalability of application protocol analyzers. In this section, we show the effectiveness of the load-balancing mechanism of the flow abstractor through various experiments. In our experiments, the HTTP analyzer was used as a heavy application-level analyzer. Experiments were executed using a PC with DDR3 1.5 TB memory and an Intel Xeon E7-4830v2 processor (10 cores, 2.2 GHz, 20 MB cache) $\times$ 4 and the Ubuntu 14.10 (Linux Kernel 3.16) operating system.

The CPU loads of the HTTP analyzer and flow abstractor when generating HTTP requests are shown in Figure 7. In the figure, 50 HTTP clients were generated per second, with a maximum of 1,000 clients; on average, 2,500 HTTP requests were generated per second. Figures 7(a), (b), and (c) show CPU loads when load balancing was executed using one, two, and four HTTP analyzer processes, respectively.

When only one HTTP analyzer process was used, it could manage approximately 2,500 requests per second because of CPU saturation. However, when two processes were used, each process consumed only approximately 50% of CPU resources (i.e., it was not saturated). Moreover, when four processes were used, only approximately 25% of CPU resources were consumed. Consequently, we conclude that the load-balancing mechanism is remarkably efficient for multicore scalability. In our experiments, although the HTTP analyzer was implemented in Python (a relatively slow interpreted language), we could completely manage C10K using four HTTP analyzer processes.

Total memory usage of the HTTP analyzer is shown in Figure 8. When executing one, two, and four processes, approximately 12, 23, and 43 MB memory were allocated to them, respectively. Consequently, we conclude that memory usage proportionally increases with the number of processes; however, it is probably sufficiently small to allow application in the real world.

5.2 Performance Evaluation of the Flow Abstractor

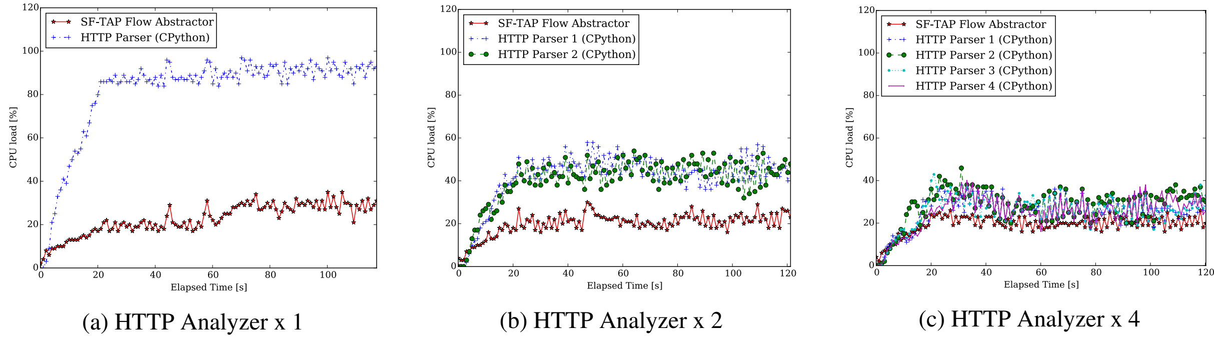
In this subsection, we show our experimental results for the flow abstractor. These experiments were conducted using the same PC described in Section 5.1.

Figure 9, in which CPS implies connections per second, shows packet-dropping rates when many TCP connections were generated. Here, one TCP session consisted of five packets: a 3-way handshake and two 1400-byte data packets. For comparison, we also determined the performances of tcpdump and Snort. We specified a 1 GB buffer for the flow abstractor and tcpdump; furthermore, we specified the maximum values possible for the elements of Snort's configuration, such as the limitation of memory usage and the number of TCP sessions. Experimental results showed that tcpdump, Snort, and the flow abstractor can manage approximately 3,200, 10,000, and 50,000 CPS, respectively. We achieved these performances because of multithreading and bulk data transfers described in Section 4.1. The flow abstractor completely separates capturing and parsing functions into different threads. Furthermore, bulk data transfers mitigated the performance overhead caused by spin lock and thread scheduling.

Figures 10(a), (b), and (c) show CPU loads when generating 1K, 5K, and 10K TCP CPS for up to 10 M connections, respectively. Because our implementation maintains TCP sessions by std::map of C++, the number of TCP connections affects the CPU load of the flow abstractor. For 10K CPS, the average CPU load exceeded 100%. This shows that the flow abstractor scales up to multiple CPU cores because of multithreading.

In Figure 10(c), after approximately 400 s, the CPU load slightly decreased from approximately 150% to 120%. This was probably caused by our garbage collection algorithm for TCP sessions. In our implementation, when the number of TCP sessions maintained by the flow abstractor is sufficiently small, the garbage collector scans all TCP sessions; on the other hand, when the number of TCP sessions is large, it partially scans TCP sessions to avoid a lock being caused by the garbage collector over a long time period.

Figure 11 shows CPU loads for traffic volumes of 1, 3, 5, and 7 Gbps. Here, we generated only one flow per measurement. Given these results, we conclude that the flow abstractor can manage high-bandwidth network



(a) HTTP Analyzer x 1 (b) HTTP Analyzer x 2 (c) HTTP Analyzer x 4

generate 50 clients / sec, 1000 clients maximum, 2500 requests / sec on average

Figure 7: CPU Load of HTTP Analyzer and Flow Abstractor

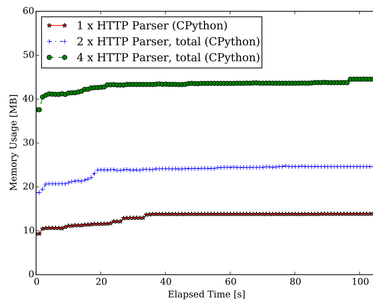


Figure 8: Total Memory Usage of HTTP Analyzer

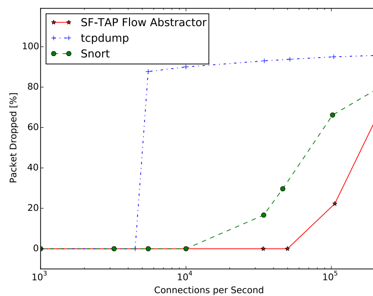


Figure 9: Packet Drop against CPS

traffic when the number of flows is small.

Figure 12 shows physical memory usage of the flow abstractor when generating 10K TCP CPS. The amount of memory usage of the flow abstractor primarily depends on the number of TCP sessions. More specifically, the amount of memory usage increases proportionally with the number of TCP sessions.

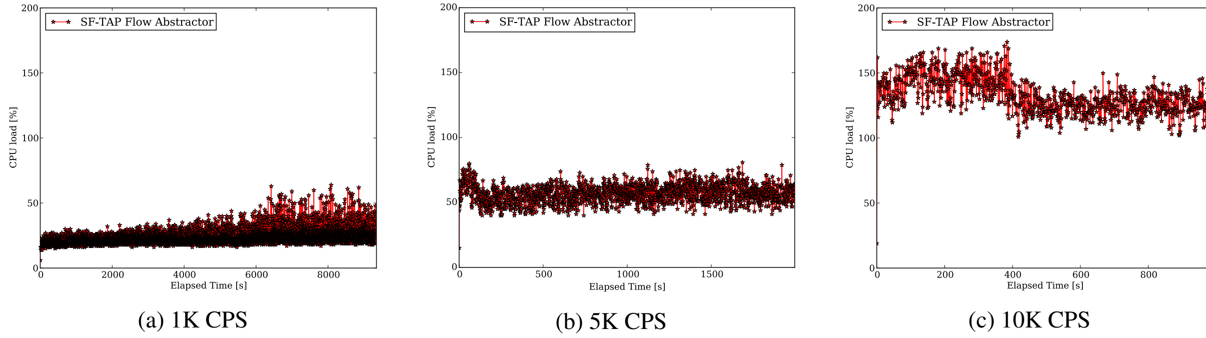
5.3 Performance Evaluation of the Cell Incubator

In the experiments involving the cell incubator, we used a PC with DDR3 16 GB Memory and an Intel Xeon E5-2470 v2 processor (10 cores, 2.4 GHz, 25 MB cache) and FreeBSD 10.1. The computer was equipped with four Intel quad-port 1 GbE NICs and an Intel dual-port 10 GbE NIC. We generated network traffic consisting of short packets (i.e., 64-byte L2 frames) on the 10 GbE lines for our evaluations. The cell incubator separated traffic based on the flows, with the separated flows forwarded to the twelve 1 GbE lines. Figure 13 shows our experimental network.

We conducted our experiments using three patterns: (1) the cell incubator worked in the mirroring mode using port mirroring on the L2 switch; in other words, it captured packets at α and forwarded packets to γ; (2) the cell incubator worked in the inline mode but did not forward packets to 1 GbE NICs, instead only α to β; and (3) the cell incubator worked in the inline mode, capturing packets at α and forwarding to both β and γ.

Table 14 shows the performance of the cell incubator. For pattern (1), i.e., the mirroring mode, the cell incubator could manage packets up to 12.49 Mpps. For pattern (2), i.e., the cell incubator working as an L2 bridge, it could forward packets up to 11.60 Mpps. For pattern (3), i.e., forwarding packets to β and γ, the cell incubator could forward packets to β and γ up to 11.44 Mpps. The performance of the inline mode was poorer than that of the mirroring mode because packets were forwarded to two NICs when using the inline mode. However, the inline mode is more suitable for specific purposes such as IDS/IPS because the same packets are dropped at β and γ. In other words, all transmitted packets can be captured when using the inline mode.

Table 15 shows the CPU load averages of the cell incubator when in the inline mode and forwarding 64-byte frames. At 5.95 and 10.42 Mpps, packets were not dropped when forwarding. At approximately 10.42 Mpps, the upper limit of dropless forwarding was reached. This indicates that several CPUs were used for forwarding, but the 15th CPU's resources were especially consumed.



(a) 1K CPS (b) 5K CPS (c) 10K CPS

generate 1K, 5K and 10K CPS, 10M connections maximum

Figure 10: CPU Loads of Flow Abstractor versus CPS

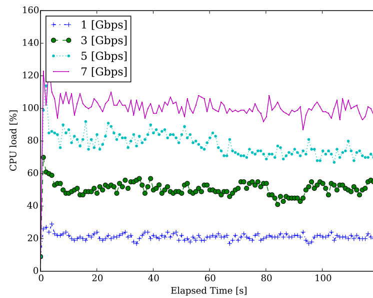


Figure 11: CPU Load of Flow Abstractor versus Traffic Volume

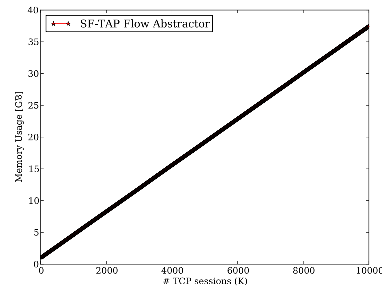


Figure 12: Physical Memory Usage of Flow Abstractor (10K CPS)

Figure 16 shows the CPU loads of the 15th CPU. At 5.95 Mpps, the load average was approximately 50%, but at 10.42 Mpps, the loads were close to 100%. Moreover, at 14.88 Mpps, CPU resources were completely consumed. This limitation in forwarding performance was probably caused by the bias, which in turn was due to the flow director [10] of Intel's NIC and its driver. The flow director cannot currently be controlled by user programs on FreeBSD; thus, it causes bias depending on network flows. Note that the fairness regarding RSS queues is simply an implementation issue and is benchmarked for future work.

Finally, the memory utilization of the cell incubator depends on the memory allocation strategy of netmap. The current implementation of the cell incubator requires approximately 700 MB of memory to conduct the experiments.

6 Discussion and Future Work

In this section, we discuss performance improvements and pervasive monitoring.

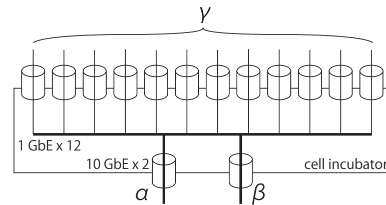


Figure 13: Experimental Network of Cell Incubator

6.1 Performance Improvements

We plan to improve the performance of the flow abstractor in three aspects.

(1) The UNIX domain socket can be replaced by another mechanism such as a memory-mapped file or cross-memory attach [6]; however, these mechanisms are not suitable for our approach, which abstracts flows as files. Thus, new mechanisms for high-performance message passing, such as the zero-copy UNIX domain socket or zero-copy pipe, should be studied.

(2) The flow abstractor currently uses the malloc function for memory allocation, which has some overhead. Here, malloc can be replaced by another lightweight

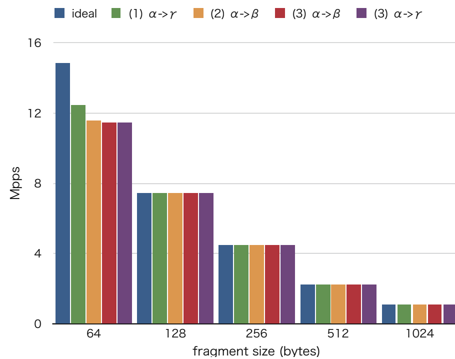


Figure 14: Forwarding Performance of Cell Incubator (10 Gbps)

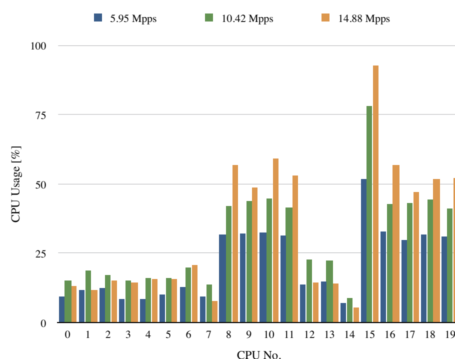


Figure 15: CPU Load Average of Cell Incubator (64-byte frames)

mechanism such as a slab allocator. The replacement of malloc by a slab allocator therefore constitutes an aspect of our future work.

(3) The flow abstractor adopts regular expressions for application protocol detection. We profiled the flow abstractor, but at present, this is not critical. Nonetheless, it potentially requires high computational resources. Thus, high-performance regular expressions should be studied in the future. Some studies have taken advantage of GPGPUs for high-performance regular expressions [5, 37, 41]. The implementation of regular expressions using GPGPUs is therefore another aspect of our future work.

6.2 Pervasive Monitoring and Countermeasures

Pervasive monitoring [9] is an important issue on the Internet. Countermeasures against pervasive monitoring include using cryptographic protocols such as SSL/TLS instead of traditional protocols such as HTTP and FTP, which are insecure. However, cryptographic protocols invalidate IDS/IPS, and consequently, other security risks are incurred.

Host-based IDS/IPS is a solution to the problem, but it is not suitable for mobile devices, which are widely used in today's society, because of the lack of machine power. Therefore, new approaches such as IDS/IPS cooperating with an SSL/TLS proxy should be studied to support the future of the Internet. The L7 loopback interface of the flow abstractor may also help future IDS/IPS implementations to be more robust against cryptographic protocols.

7 Related Work

Wireshark [38] and tcpdump [35] are widely used traditional packet-capturing applications, and libnids [15] is a network traffic-capturing application that reassembles TCP streams. The execution of these applications is essentially single threaded. Thus, they do not take advantage of multiple CPU cores and are therefore not suitable for high-bandwidth network traffic analysis.

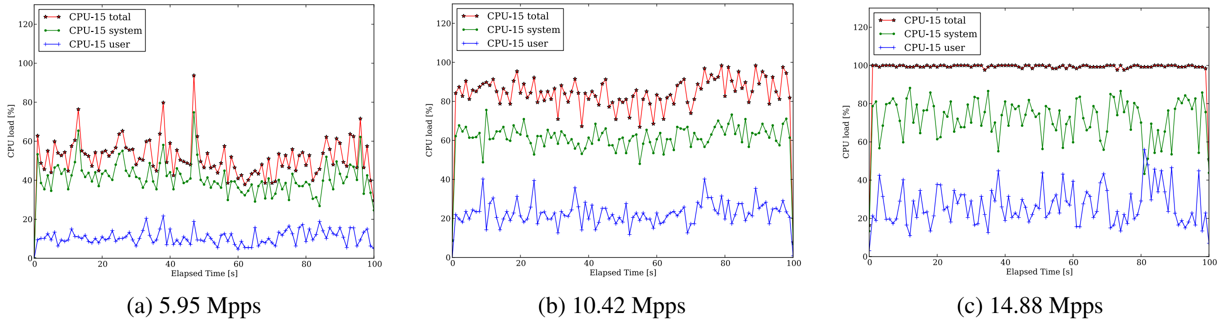
SCAP [25] and GASPP [36] were proposed for flow-level and high-bandwidth network traffic analyses. SCAP is implemented within a Linux kernel, taking advantage of the zero-copy mechanism and allocating threads for NIC's RX and TX queues to achieve high throughput. In addition, SCAP adopts a mechanism called subzero-copy packet transfer using analyzers that can selectively analyze required network traffic. GASPP is a GPGPU-based flow-level analysis engine that uses netmap [30]; thus, GASPP achieves high-throughput data transfers between the NIC and CPU memory.

DPDK [8], netmap [30], and PF_RING [27] were proposed for high-bandwidth packet-capture implementations. In traditional methods, many data transfers and software interrupts occur among the NIC, kernel, and user, thus making it difficult to capture 10 Gbps network traffic using traditional methods. Our proposed method achieved wire-speed traffic capture by effectively reducing the frequency of memory copies and software interrupts.

L7 filter [13], nDPI [20], libprotoident [16], and PEAFLOW [7] have been proposed for application-level network traffic classification implementations. These methods use Aho–Corasick or regular expressions to detect application protocols. PEAFLOW uses a parallel programming language called FastFlow to achieve high-performance classification.

IDS applications such as Snort [33], Bro [4], and Suricata [34] reconstruct TCP flows and application-level analysis. BinPAC [24] is a DSL used by Bro for protocol parsing; however, Snort and Bro are single threaded and cannot manage high-bandwidth network traffic. On the other hand, Suricata is multithreaded and manages high-bandwidth network traffic.

Schneider et al. [31] proposed a horizontally scalable



(a) 5.95 Mpps (b) 10.42 Mpps (c) 14.88 Mpps

Figure 16: 15th CPU's Load of Cell Incubator (64-byte frames)

architecture that separates 10 Gbps based on the flows, much like SF-TAP. They verified their architecture using only 1 Gbps network traffic and are yet to verify it using 10 Gbps network traffic.

Open vSwitch [23] is a software switch that can control network traffic based on the flows, much like our cell incubator does in our system, but its OVF CTRL cannot manage IP fragmentation. Some filtering mechanisms such as iptables [11] and pf [26] can also control network traffic based on the flows, but these mechanisms cannot manage IP fragmentation. Furthermore, these methods are less scalable and characterized by performance issues.

In Click [12], SwitchBlade [1], and ServerSwitch [17], modular architectures were adopted to provide flexible and programmable network functions for network switches. In SF-TAP, we adopted these ideas and proposed a modular architecture for network traffic analysis.

BPF [19] is a well-known mechanism for packet capturing that abstracts network traffic as files, much like UNIX's/dev. In SF-TAP, we adopted this idea, abstracting network flows as files to achieve modularity and multicore scaling.

8 Conclusion

Application-level network traffic analysis and sophisticated analysis techniques such as machine learning and stream data processing for network traffic require considerable computational resources. Therefore, in this paper, we proposed a scalable and flexible traffic analysis platform called SF-TAP for sophisticated high-bandwidth real-time application-level network traffic analysis.

SF-TAP consists of four planes: the separator plane, abstractor plane, capturer plane, and analyzer plane. First, network traffic is captured at the capturer plane, and then, captured network traffic is separated based on the flows at the separator plane, thus achieving horizontal scalability. Separated network traffic is forwarded to multiple SF-TAP cells, which consist of the abstractor and analyzer planes.

We provided cell incubator and flow abstractor implementations for the separator and abstractor planes, respectively. Furthermore, we implemented an HTTP analyzer as an example analyzer at the analyzer plane. The capturer plane adopts well-known technologies, such as port mirroring of L2 switches, for traffic capturing.

The flow abstractor abstracts network traffic into files, much like Plan9, UNIX's /dev, and BPF; the architecture of the flow abstractor is modular. The abstraction and modularity allow application-level analyzers to be easily developed in many programming languages and be multicore scalable. We showed experimentally that the HTTP analyzer we implemented as an example using Python can be easily scaled to multiple CPU cores.

The flow abstractor takes advantage of multithreading and bulk data transfers among threads. Thus, from our experiments, we found that the flow abstractor can manage up to 50K connections per second without dropping packets; tcpdump and Snort can manage only up to 4K and 10K connections per second, respectively.

In addition, we showed that the flow abstraction interfaces can help scale the HTTP analyzer to multiple CPU cores. Our experiments showed that our HTTP analyzer written in Python as a single process consumed 100% of CPU resources, but with four processes, each process only consumed 25% of CPU resources.

The cell incubator is a component that provides horizontal scalability. To manage high-bandwidth network traffic, the cell incubator separates network traffic based on the flows, forwarding separated flows to cells that consist of a flow abstractor and application-level analyzers. We experimentally showed that the cell incubator can manage approximately 12.49 Mpps and 11.44 Mpps when in the mirroring and inline modes, respectively.

Acknowledgments

We would like to thank the staff of StarBED for supporting our research and WIDE Project for supporting our experiments.

References

[1] Anwer, M. B., Motiwala, M., Tariq, M. M. B., and Feamster, N. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010* (2010), S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, Eds., ACM, pp. 183–194.

[2] BitTorrent. http://www.bittorrent.com/.

[3] Boost C++ Library. http://www.boost.org/.

[4] The Bro Network Security Monitor. http://www.bro.org/.

[5] Cascarano, N., Rolando, P., Risso, F., and Sisto, R. iNFAnt: NFA pattern matching on GPGPU devices. *Computer Communication Review 40*, 5 (2010), 20–26.

[6] Cross Memory Attach (index : kernel/git/torvalds/linux.git). http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=fcf634098c00dd9cd247447368495f0b79be12d1.

[7] Danelutto, M., Deri, L., Sensi, D. D., and Torquati, M. Deep Packet Inspection on Commodity Hardware using FastFlow. In *PARCO* (2013), M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, Eds., vol. 25 of *Advances in Parallel Computing*, IOS Press, pp. 92–99.

[8] Intel@ DPDK: Data Plane Development Kit. http://www.ntop.org/products/pf_ring/.

[9] Farrell, S., and Tschofenig, H. Pervasive Monitoring Is an Attack. RFC 7258 (Best Current Practice), May 2014.

[10] Intel®, High Peformance Packet Processing. https://networkbuilders.intel.com/docs/network_builders_RA_packet_processing.pdf.

[11] netfilter/iptables project homepage - The netfilter.org "iptables" project. http://www.netfilter.org/projects/iptables/.

[12] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[13] L7-filter — ClearFoundation. http://l7-filter.clearfoundation.com/.

[14] libevent. http://libevent.org/.

[15] libnids. http://libnids.sourceforge.net/.

[16] WAND Network Research Group: libprotoident. http://research.wand.net.nz/software/libprotoident.php.

[17] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011* (2011), D. G. Andersen and S. Ratnasamy, Eds., USENIX Association, pp. 15–28.

[18] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V. A., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 459–473.

[19] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993* (1993), USENIX Association, pp. 259–270.

[20] nDPI. http://www.ntop.org/products/ndpi/.

[21] The netmap project. http://info.iet.unipi.it/~luigi/netmap/.

[22] Leading operators create ETSI standards group for network functions virtualization. http://www.etsi.org/index.php/news-events/news/644-2013-01-isg-nfv-created.

[23] Open vSwitch. http://openvswitch.github.io/.

[24] Pang, R., Paxson, V., Sommer, R., and Peterson, L. L. binpac: a yacc for writing application protocol parsers. In *Internet Measurement Conference* (2006), J. M. Almeida, V. A. F. Almeida, and P. Barford, Eds., ACM, pp. 289–300.

[25] Papadogiannakis, A., Polychronakis, M., and Markatos, E. P. Scap: stream-oriented network traffic capture and analysis for high-speed networks. In *Internet Measurement Conference, IMC'13, Barcelona, Spain, October 23-25, 2013* (2013), K. Papagiannaki, P. K. Gummadi, and C. Partridge, Eds., ACM, pp. 441–454.

[26] PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/.

[27] PF_RING. http://www.ntop.org/products/pf_ring/.

[28] Pike, R., Presotto, D. L., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan 9 from bell labs. *Computing Systems 8*, 2 (1995), 221–254.

[29] RE2. https://github.com/google/re2.

[30] Rizzo, L., and Landi, M. netmap: memory mapped access to network devices. In *SIGCOMM* (2011), S. Keshav, J. Liebeherr, J. W. Byers, and J. C. Mogul, Eds., ACM, pp. 422–423.

[31] Schneider, F., Wallerich, J., and Feldmann, A. Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware. In *PAM* (2007), S. Uhlig, K. Papagiannaki, and O. Bonaventure, Eds., vol. 4427 of *Lecture Notes in Computer Science*, Springer, pp. 207–217.

[32] SF-TAP: Scalable and Flexible Traffic Analysis Platform. https://github.com/SF-TAP.

[33] Snort :: Home Page. https://www.snort.org/.

[34] Suricata — Open Source IDS / IPS / NSM engine. http://suricata-ids.org/.

[35] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/.

[36] Vasiliadis, G., Koromilas, L., Polychronakis, M., and Ioannidis, S. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.* (2014), G. Gibson and N. Zeldovich, Eds., USENIX Association, pp. 321–332.

[37] Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E. P., and Ioannidis, S. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings* (2009), E. Kirda, S. Jha, and D. Balzarotti, Eds., vol. 5758 of *Lecture Notes in Computer Science*, Springer, pp. 265–283.

[38] Wireshark - Go Deep. https://www.wireshark.org/.

[39] The Official YAML Web Site. http://yaml.org/.

[40] A YAML parser and emitter in C++. https://github.com/jbeder/yaml-cpp.

[41] Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., and Dong, Q. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012* (2012), J. Ramanujam and P. Sadayappan, Eds., ACM, pp. 129–140.

Spyglass: Demand-Provisioned Linux Containers for Private Network Access

Patrick T. Cable II, Nabil Schear
MIT Lincoln Laboratory
{cable,nabil}@ll.mit.edu

Abstract

System administrators are required to access the privileged, or "super-user," interfaces of computing, networking, and storage resources they support. This low-level infrastructure underpins most of the security tools and features common today and is assumed to be secure. A malicious system administrator or malware on the system administrator's client system can silently subvert this computing infrastructure. In the case of cloud system administrators, unauthorized privileged access has the potential to cause grave damage to the cloud provider and their customers. In this paper, we describe Spyglass, a tool for managing, securing, and auditing administrator access to private or sensitive infrastructure networks by creating on-demand bastion hosts inside of Linux containers. These on-demand bastion containers differ from regular bastion hosts in that they are nonpersistent and last only for the duration of the administrator's access. Spyglass also captures command input and screen output of all administrator activities from outside the container, allowing monitoring of sensitive infrastructure and understanding of the actions of an adversary in the event of a compromise. Through our evaluation of Spyglass for remote network access, we show that it is more difficult to penetrate than existing solutions, does not introduce delays or major workflow changes, and increases the amount of tamper-resistant auditing information that is captured about a system administrator's access.

1 Introduction

System administrators have super-user access to the low-level infrastructure of the systems and networks they maintain. To effectively do their job, they need to access the sensitive interfaces of switches, routers, operating systems, firmware, virtualization platforms, security appliances, etc. We rely increasingly on this infrastructure for tasks, from ordering food to controlling complex mechanical systems like the electric grid. Given the typical administrator's breadth of access to this infrastructure, administrators or the client devices they use are a prime target for compromise by a motivated adversary. Alternatively, if the administrator and the adversary are the same (i.e., a rogue administrator or insider), then this administrator often has unchecked access to disable and evade the security controls of the network.

To protect the sensitive interfaces an administrator must use, the system architect can place these interfaces on a private network or VLAN that is not broadly accessible to either the Internet or even and organizational LAN. This practice is also commonplace in Infrastructure-as-a-Service cloud environments at both the tenant layer (e.g., the user of virtual machines) and the provider layer (e.g., the operator of the virtual machine hosting environment) [27]. Firewalls, virtual private networks (VPNs), and bastion hosts allow remote access for the administrator into the sensitive network. Firewalls and VPNs open new security vulnerabilities by directly connecting a potentially untrusted client system directly to the sensitive network, and the do not directly offer an audit log of the administrator's activities. Bastion hosts explicitly isolate the client system from the network and offer a centralized place to audit activities. However, bastion hosts themselves can be compromised, leading to a catastrophic security collapse where the adversary can impersonate *any* administrator and wreak havoc across the network.

To address the security shortcomings of bastion hosts, while retaining good network isolation and audit capabilities, we created Spyglass. Spyglass is a tool that provides on-demand nonpersistent bastion hosts to each administrator to facilitate access to sensitive networks. The

This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

system creates a Linux container using Docker for each user's session and destroys it after the user disconnects. Through a least-privilege system design, Spyglass lowers the risk of compromise to the bastion server itself. While Spyglass does not prevent insiders with valid credentials from accessing the sensitive network, it does provide a tamper-resistant audit record of their activities. This capability allows an organization to forensically track the moves of adversaries and assists in recovery and cleanup.

This paper's primary contributions are:

- Design for securely isolating and monitoring the actions of system administrators while reducing the threat posed by insiders and phishing attacks
- Implementation of the Spyglass prototype and best practices for deployment
- Security and performance evaluation showing that Spyglass is more difficult to penetrate than previous solutions and that it can be implemented without considerable delay or workflow changes.

The rest of this paper is structured as follows: Section 2 describes the the problem, threat model, and existing solutions. Section 3 discusses the design of the system. In Section 4, we describe the components of the system and their implementations. We evaluate both the performance and security in Section 5. Section 6 reviews related work. We discuss the current status of Spyglass and opportunities for future work in Section 7, and conclude in Section 8.

2 Background

A system administrator often connects to a variety of interfaces to perform their work. These interfaces may be used to configure switching or routing logic, or to access hardware in the event of a system crash. The administrator may connect to the host running a virtualization platform, or a machine instance operating a cloud platform. Given the success and prevalence of DevOps environments, the administrator may also be making code changes to the software that actually runs the provider's self-service platform.

It is easy to see why administrator credentials are so sought by adversaries, either those looking to compromise an administrator for an organization, or a software-as-a-service customer of that organization. Credential theft can be crippling: in June 2014, an adversary compromised the Amazon Web Services (AWS) credentials of CodeSpaces, a company that provided cloud-based source code repository hosting. The adversary then asked the company for a sum of money by a certain time. When the money was not paid, the adversary then deleted all of CodeSpaces' AWS instances and disk storage, along with all of their backups. The company folded shortly thereafter [10].

One of the most popular ways to obtain credentials is by phishing. In the most damaging phishing attacks, an adversary convinces an administrator to install malware on their computer, steals their credentials, and then spreads across the network that the administrator maintains. Some of the most serious breaches of 2014, including those on Sony Pictures [3] and JP Morgan Chase Bank [2], involved the theft and misuse of administrative credentials. Indeed, these attacks were most damaging precisely because of this fact.

Given that our infrastructure can be compromised by either an inside or outside adversary, we need a solution to limit the impact of these attacks. As part of security best practices, the networks on which the most sensitive of these interfaces are hosted are often separated from public-facing or even internal LANs. This limits the accessibility of these sensitive interfaces and protects the credentials for accessing them from eavesdropping.

Since administrators must invariably access these isolated networks to do their work, we need ways to facilitate remote access. The goals of an ideal remote access solution should provide security for remote access, strong authentication, and audit logging of all actions that take place across the trust boundary. In the following sections, we describe the existing remote access solutions and their strengths and weaknesses with respect to this set of goals.

2.1 Firewalls

When a sensitive network is firewalled off from an untrusted network, the firewall allows or denies traffic based on policy rules. This provides a layer of security to the protected network. Hosts exposed to external networks need to contend with malicious traffic, many of which attempt to brute-force common passwords or attempt known attacks en masse to any host that will listen. The firewall allows for a central focus point for security decisions, and enforcement of security policy [37]. Indeed, "firewalls are an important tool that can minimize the danger, while providing most – but not necessarily all – of the benefits of a network connection" [1].

The downside of firewalling traffic is that it only allows network-based filtering of traffic. *Firewalls* do not establish authorization of a user to connect. They do not protect against IP spoofing attacks. Multiple users could be behind an IP address that is chosen as an appropriate host from which to receive traffic. This leaves the authorization decision to the remote device. Firewalls also do not protect against a trusted insider. Similarly, they do not do anything for the remote host in the sensitive

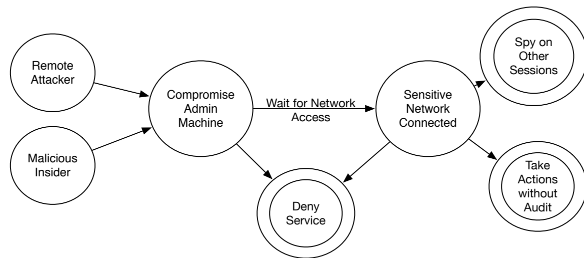


Figure 1: A state diagram showing steps required to compromise a sensitive network protected with Firewalls and VPNs.

network:

> Given that the target of the attackers is the hosts on the network, should they not be suitably configured and armored to resist attack? The answer is that they should be, but probably cannot. Such attempts are probably futile. There will be bugs, either in the network programs or in the administration of the system. [1]

Indeed, there are bugs. For example, many administrators use the Intelligent Platform Management Interface (IPMI) to perform remote administration functions. This protocol and the hardware that implements it are both critical to the ability to remotely debug system failures and problems. Yet, one independent researcher found that close to 90% of implementations that were publicly accessible had a security issue that would allow unauthorized access to the hardware [7]. Some of these expose their password by querying a device using Telnet [34].

We show a state diagram in Figure 1 that illustrates what an attacker would have to do to compromise the sensitive network. A firewalled network may always be connected to the host. This reduces the amount of time an adversary may have to wait to compromise the sensitive network.

2.2 Virtual Private Networks

A popular methodology for separating sensitive and untrusted networks is to place a host between two networks. In the VPN methodology, the host runs software such as OpenVPN that facilitates a remote host "joining" the network as if it were there locally [25]. Many organizations use this to facilitate remote workers: the worker can be anywhere, and the traffic between the company and the end user's laptop is encrypted to prevent the data from interception or eavesdropping.

The ubiquity of the VPN is due in part to its ease of use. A user installs a client application configured by their organization, and is able to connect to the network and access the network in its entirety. Applications don't need to be redesigned to deal with external access, and an organization can rest assured that most of their data stays on the internal network.

However, in the era of the "French-bread model" of network security, this has meant that an external laptop has unfettered access to the soft inside of the network [13]. This makes the administrator's laptop a perfect pivot point to infiltrate a network that connects to sensitive infrastructure. Many organizations attempt to deal with this risk via policy. For example, policies like "Establish a VPN connection immediately after establishing Internet connectivity" and "do not connect any non-work-owned devices to a work-owned laptop" are common. There are multiple reasons, intentional and unintentional, that may cause an employee to not follow the rules. For example, an employee may connect to a malicious wireless access point. The owner of the malicious access point may inject advertisements that are provided by a malware carrier, infecting the computer.

Finally, while logging and auditing of VPN connections themselves is straightforward to implement at the VPN concentrator, correlation of a user activities through network logs, host logs, and authentication information is more challenging. First, the VPN connection will virtually connect the remote user to a dynamically chosen IP address within the sensitive network that may have previously been used by another VPN user. Second, the VPN user's activities on the host (e.g., commands executed or data copied) must be combined with network logs to understand the impact of a malicious actor.

2.3 Bastion Hosts

Bastion hosts are like the lobby of a building: "Outsiders may not be able to go up the stairs and may not be able to get into the elevators, but they can walk freely into the lobby and ask for what they want" [37]. Bastion hosts provide a single point to *audit* traffic as an interface that can be controlled by the organization that owns or controls the private network, as opposed to just being able to see basic network flow data (source, destination, session duration, etc.). Firewalls and VPNs allow you direct access to a remote network, without having to necessarily "check in."

Providing a controlled interactive session, as opposed to firewalled or VPN-based access, carries benefits for the organization that controls the sensitive network. The organization does not have to worry *as much* about the state of the administrator's workstation. The organization can employ software and methods used to monitor workstations and integrate these with existing security infrastructure. Figure 2 shows that compromise of the sensitive network is more difficult with a bastion host than with firewalls or VPNs.

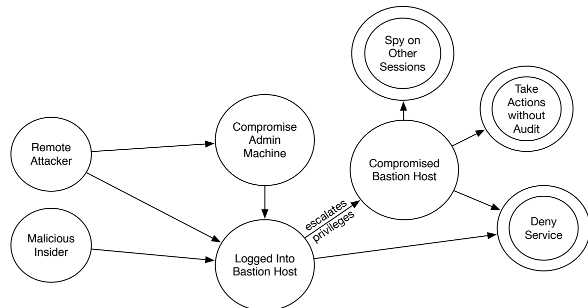


Figure 2: A state diagram showing steps required to compromise a sensitive network protected with a traditional bastion host.

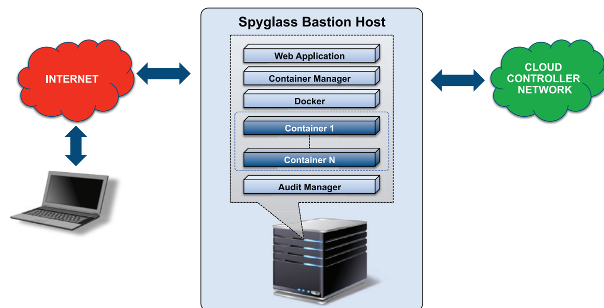


Figure 3: Spyglass System Design

Bastion hosts provide a centralized point at which to enforce strong authentication and to capture detailed audit logs of user activity. Their primary weakness is in the new vulnerabilities they introduce. The act of providing an interactive session on the bastion host to an end user is risky. In the case of a malicious insider, the organization has given logical access to a bastion host; if any pieces of software on the bastion host are compromised, the insider can attribute actions to other users, get a set of password hashes of other accounts, and/or key log to gain access to other devices on the sensitive network. Literature going back decades covers how attackers break into bastion hosts and create persistent environments [4].

3 Design

We believe the bastion host pattern provides the best solution to achieve secure and audible remote access for system administrators. To implement a secure bastion host, we need to address the weaknesses in typical bastion deployments like single point of failure, tamperable audit information, and weak passwords.

Our goals in this work are to minimize the risk of the bastion itself, while providing higher security for system administrators and the isolated networks they use. We want to have the ability to audit and log, in a tamper-resistant manner, all activities that a user makes on the sensitive network. We want to limit the spread of an external attacker and the impact they can have. Finally, we want to ensure that even if they do compromise the sensitive network, we can recover using the audit log.

To address these challenges and our set of goals, we developed Spyglass. Spyglass is a network access device that is dual homed on an untrusted network and sensitive network where the interfaces to critical security infrastructure reside (see Figure 3). A user wishing to access the sensitive network authenticates to Spyglass and requests a nonpersistent, isolated session. From this session, the user can access resources on the sensitive network. From a vantage point outside of the user's session, Spyglass monitors and records all of the user's activities. In the following sections we review the threat model for Spyglass and then discuss its four key design components: multifactor authentication, isolation, nonpersistence, and auditing.

3.1 Threat Model

We assume that the adversary is either a malicious system administrator or that the system administrator's client system has been compromised. The goal of the adversary is to compromise an isolated network the system administrator controls. We assume that the adversary may be able to compromise applications inside of the containers that face the untrusted network. However, we assume the adversary cannot break out of the sandbox Spyglass creates, and cannot compromise the control process for creating and destroying containers. We assume that valid users have used multiple factors to authenticate to the system creating the containers. Finally, we assume that SSH is properly configured (i.e., disabling tunneling) along with the network bridge device used by the OS-level virtualization provider.

3.2 Multifactor Authentication

We begin by considering how to authenticate Spyglass users. Reusable passwords are both easily cracked and easily stolen [21]. Indeed, with custom hardware, an adversary can crack passwords at a rate of 350 billion guesses per second [9]. Adding multiple factors makes it more difficult to steal a user's credentials to obtain unauthorized access. Some types of multifactor authentication require that the valid user be physically present to initiate a session. In the case of an administrator whose client system is compromised by an external attacker, this slows the attacker to only be able to initiate a session when they can subvert one initiated by the valid user.

Best practices for organizational cyber security also agree on the importance of multifactor authentication. For example, the SANS Institute's Critical Security Con-

trols recommend that multifactor authentication should be used "for all administrative access, including domain administrative access" [30]. Unfortunately, some assets (e.g., networking or storage appliances) cannot take advantage of multifactor authentication natively. By introducing the Spyglass bastion in front of these systems, we are able to better address the SANS control's recommendation.

Many government institutions implement multifactor authentication by taking advantage of the cryptographic functions of smart cards. For organizations that already have the infrastructure required to operate a large smart card infrastructure, this could be suitable for Spyglass. However, the implementation often requires additional hardware (both in the form of the cards and readers). It also requires complicated integration to allow for those capabilities to be used to authenticate with websites.

To address these scalability and adoption challenges, we chose Yubico YubiKey to add another authentication factor for Spyglass authentication [36]. The YubiKey is a USB device that outputs a 44-character string of ASCII letters that represent a 12-character identifier and a 32-character one-time password based on the secret and public identifier stored on the hardware device. A large community exists around the use of the YubiKey, and an open-source YubiKey Validation Server exists along with cross-language libraries to interface with the server.

3.3 Isolation

While it's certainly easier for a user to directly connect to a sensitive network (either via firewall or via VPN), as discussed in Sections 2.1 and 2.2, it comes at a cost to the security posture of the sensitive network. Malware on the users system may have unfettered access to the sensitive network and may directly connect to and attack hosts there. For this reason, Spyglass, as other bastion hosts do, explicitly isolates and separates the administrator's computer and the sensitive network. Isolation is critical, since "[t]oday's cyber incidents result directly from connecting formerly standalone or private systems and applications to the Internet and partner networks" [8].

We also introduce isolation between the different users of Spyglass and the components that underpin Spyglass. Thus, each user gets their own login environment from which to pivot to the sensitive network and, similarly, each component of Spyglass is in an isolated environment and only communicates to other components over minimal well-defined interfaces. Traditionally, virtualization provides an answer for system architectures that required that two subsystems couldn't necessarily affect each other's memory space in unexpected ways. However, that assurance comes at a performance cost. Creating a virtual machine for each user would pose a significant resource overhead and delay considering that virtual machine spin-up times (even in the cloud) exceed 30 seconds regularly.

To achieve strong isolation without the performance overhead of full system virtualization, we utilize OS-level virtualization technologies to isolate Spyglass components and users. This method of virtualization allows multiple environments to share a common host kernel and utilize underlying OS interfaces, thus incurring less CPU, memory, and networking overload [29].

3.4 Nonpersistence

Increasing the amount of ephemerality in the system design works to the organization's advantage in defending their systems. Goldman found the benefits that nonpersistence provides makes an attacker's job more difficult. Specifically, consider the ability it allows organizations to stand up and tear down a particular capability (in our case, remote access) in an on-demand fashion, and the ability to ensure that a particular state is regularly patched [8].

To understand the importance of nonpersistence, we need to understand the kill chain of an attack. The Cyber Kill Chain describes the steps an attacker must take to compromise a computer system. Generally, to launch a successful attack on a system, an attacker must collect useful information about a target, attempt to access the target, exploit a vulnerability for the target, launch the attack, and then find a way to maintain access to the system [24]. Nonpersistence interrupts an attacker's ability to persist by forcing session timeouts and subsequent destruction of their environment.

In Spyglass, new user sessions are always instantiated inside of a fresh container that is patched regularly. Even if an attacker can compromise the container, they will have to repeat this process regularly and potentially raise their profile in other monitoring and logging capabilities of the system, leading to a higher chance that an attacker will be detected.

3.5 Audit

The presence of some form of situational awareness when it comes to running a server that is available on an untrusted network like the Internet is an important asset. It is otherwise impossible to know whether a compromise has occurred if there isn't a means to audit and monitor accesses, user actions, and other items of interest. Considering that it is not a matter of *if* one gets hacked but rather *when* [37], it makes sense that seeing an attacker's actions that allowed them to compromise the host would aid in repair and recovery.

While often given less importance than active security measures like strong passwords or antivirus, best practices include the need for audit logging. The Australian Signals Directorate recommends "centralised and time-synchronised logging of successful and failed computer events" and "centralised and time-synchronised logging of allowed and blocked network activity" as part of their *Strategies to Mitigate Targeted Cyber Intrusions* report. Specifically:

> Centralised and time-synchronised logging and timely log analysis will increase an organisation's ability to rapidly identify patterns of suspicious behaviour and correlate logged events across multiple workstations and servers, as well as enabling easier and more effective investigation and auditing if a cyber intrusion occurs. [5]

Similarly, MITRE's report *Building Secure, Resilient Architectures for Cyber Mission Assurance* specifies detection and monitoring as one of five objectives that help achieve architecture resilience:

> While we cannot always detect advanced exploitations, we can improve our capabilities and continue to extend them on the basis of after-the-fact forensic analysis. Recognizing degradations, faults, intrusions, etc., or observing changes or compromises can become a trigger to invoke contingency procedures and strategies. [8]

These ideas lead to the requirement that a system be in place that captures all commands issued and their output for later retrieval and review. These logs need to be located on a remote host to preserve their content in the event that the bastion host machine is compromised. In Spyglass, we further protect the logs from tampering by capturing and transmitting the audit log information from outside of the user's container. Furthermore, if the attacker is able to disrupt the logging process somehow, the session is immediately terminated. This leads to a system where a system administrator is unable to take any actions on the sensitive network without leaving a trail of what they did.

4 Implementation

Spyglass consists of four components: a locally hosted YubiKey validation server, the Spyglass web interface, the container daemon, and the audit daemon. To maintain proper segmentation, the YubiKey Validation Server and the database server should be on a separate VLAN that is not on the sensitive network pictured above. Another independent host on a separate VLAN should store audit log data. Additionally, this audit host should be controlled/maintained by parties other than the system administrators using Spyglass (e.g., by a security policy or oversight rather than IT organization) to avoid the possibility of audit log tampering.

4.1 User Facing Interface

The user interface for Spyglass is required to:

- Authenticate a user
- Store a valid SSH public key for each user
- Instantiate a bastion container
- Destroy a bastion container
- Be accessible from a variety of client platforms

Figure 4 displays the Spyglass architecture, along with numbers representing relevant communication flows. During session initiation, a user (1) accesses the web application. The application (2) checks the authentication against a database and validates the other factor. Once the user is logged in, they (3) request a container and the web application sends a request to the container daemon with information about the user and the preferred key. The container daemon pulls this information from the database in step (4), and sends this information to Docker in step (5). Docker then (6) creates the container and sends information back to the web application to inform the user what host their container is running on. Finally, the user logs into their container in step (7). This creates a set of log files, which are read by the audit daemon and moved to the audit host (8). Processes that run on the bastion host are outlined with a dotted line.

Upon initial login, Spyglass presents users with the main interface in Figure 5. The user then adds an SSH public key by going to the *Keys* menu and clicking *New Key*, as seen in Figure 6. Once the key is added, the user can now start a session by going to the *Sessions* menu and clicking *New Session* as seen in Figure 7. Afterwards, Spyglass presents the user with session information (Figure 8). The user can now initiate an SSH connection to the bastion container and access the sensitive network.

4.2 Container Daemon

We need to enable the web UI to handle container management through Docker. Rather than doing this directly from the web application, we chose to implement a middleware process called the container daemon (or `containerd`). The primary motivation for this design was to avoid giving the Spyglass web application root privileges so that it could access the Docker control

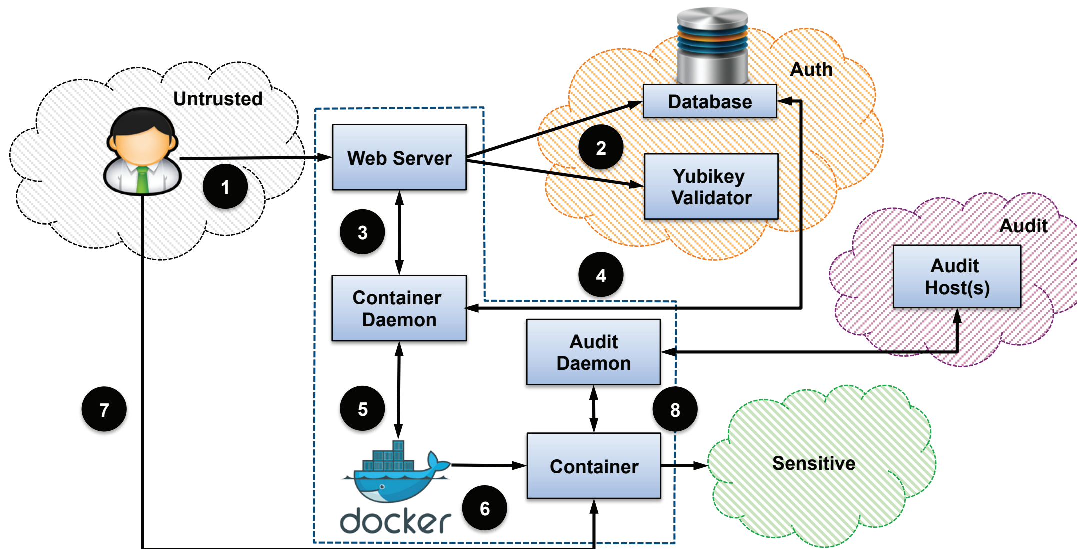


Figure 4: Spyglass infrastructure and information flows.

socket (owned by `root`). The separation also enables us, in future work, to further isolate the web application from containerd by placing the web application itself inside of a container and using SELinux mandatory access controls to implement least privilege.

We wrote containerd in a strongly typed language (Go) to prevent a variety of simple attacks on the web application itself. Containerd supports a small, simple, and well-defined interface for commands that further limits its attack surface. Thus, even if the web application is compromised, it will only able to create and delete containers and not access any other user's session.

The container daemon provides two HTTP Endpoints: `containercreate` and `containerdestroy`. The `conatinercreate` endpoint handles the creation of a container via an HTTP POST. It expects to receive a JSON object that references a database that the container request application uses.

Listing 1: A sample JSON container created notification

```
{
  "DbKeyId":10,
  "DbUserId":2,
  "SshKey":"ssh-rsa AAAAB3NzaC1yc2EAA<truncated>",
  "SshUser":"cable",
  "SshPort":"49154",
  "DockerId":"c46a32bd3347<truncated>"
}
```

Upon receiving the request, `containerd` queries the request application's database to get the appropriate username and SSH public key needed to insert into the container. Once it has the appropriate metadata, containerd instantiates a `docker` object with the appropriate configuration for the container. After Docker creates and starts the container, containerd returns a JSON object with the information about the container to the request application (as Listing 1 shows).

The `containerdelete` endpoint handles the deletion of a container via an HTTP DELETE. It only accepts a container identifier, and passes it to Docker for deletion.

4.3 Audit Daemon

Initially, we evaluated SSLsnoop for use in the system to capture activity inside the bastion containers from a vantage point outside the container boundary [20]. SSLsnoop locates the SSH session keys in the SSH process memory and does real-time decryption of traffic between two hosts. However, later versions of SSH have changed the format of in-memory structures, causing SSLsnoop to be unable to proper locate the key and encrypted stream. SSLsnoop also only monitored the SSH connections originating from the bastion itself, so an attacker intent on breaking the container would go undetected.

To keep as much of the monitoring infrastructure outside of the container as possible, we settled on a hybrid solution using SudoSH and a custom log monitor. SudoSH works by spawning the user's shell inside of an environment that is transparently capturing keystrokes and screen output [12]. We are able to look into the container host's file system and use Linux's `inotify` functionality to read the logs from the container host and relay them to the audit host [23]. The `Audit Daemon` ensures that the logs are sent to the audit host regularly and thus avoid any tampering from compromise of a Spyglass container. To further ensure the integrity of the auditing system, Spy-

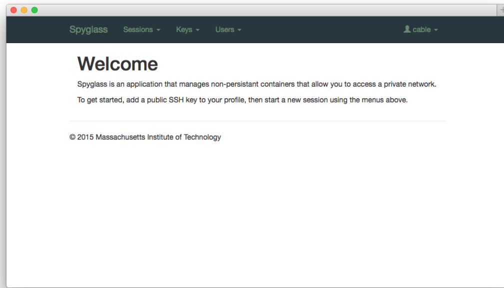


Figure 5: Initial login screen.

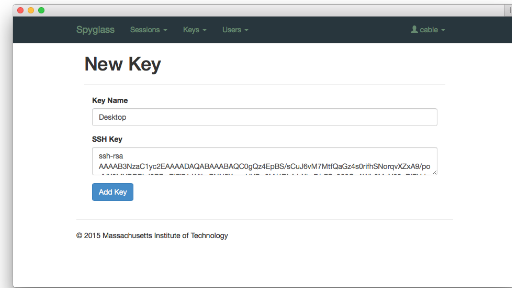


Figure 6: Adding an SSH key.

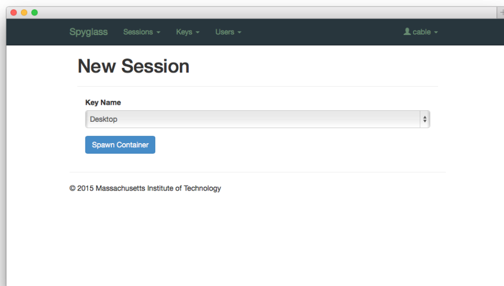


Figure 7: Creating a session.

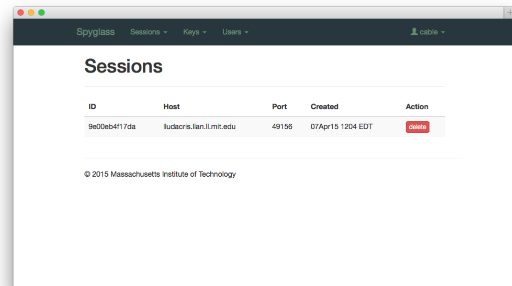


Figure 8: Viewing active sessions.

glass monitors the process table and will immediately terminate the user's bastion container if the SudoSH process stops.

5 Evaluation

We hypothesize that fast, on-demand provisioning of Linux containers that are unique to a particular user's session loosens the coupling between the integrity of the private network and the integrity of the remote client that connects to it. This separation is easy to provide as a service in part due to the lightweight nature of containers.

To prove this point, we analyzed the individual overhead of five containers on the host machine. We also attempt attacks on the system and attempt to connect to the authorization and auditing networks, along with some attempts to evade of the audit logging process. In Figure 9 we also created a state diagram, similar to Figures 1 and 2 to illustrate how Spyglass differs from existing firewall, VPN, and bastion host solutions.

These experiments were performed in VMware Fusion 7 Pro running on a Macbook Pro with 16 GB of RAM and a 2.6 GHz Intel Core i7 processor. The bastion host virtual machine has one processor core, and 1024 MB of memory.

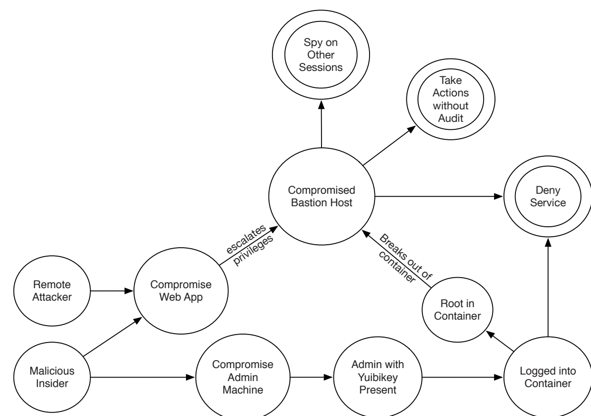


Figure 9: A state diagram showing steps required to compromise a sensitive network protected with Spyglass

5.1 Bastion Container Performance

To measure the load characteristics of an individual container, we used the Google tool cAdvisor [11] running on the bastion host. cAdvisor captures CPU and Memory load and writes it to InfluxDB, a time series database [17]. We monitored five invocations of the container that the container request application would instantiate. This session connected to a remote host and

ran the `top` command. We used the `time` command to measure the instantiation time of an individual container. We queried InfluxDB for `max(memory_usage)` and `last(cpu_cumulative_usage)` values for each individual container. Table 1 shows the results.

As expected, we find overhead and instantiation latency to be substantially lower than a virtual machine-based approach, where memory, instantiation, and CPU overhead are much larger [15]. We expect that even an embedded system with a low amount of RAM could support 10s of users. As a control channel for a sensitive network, we do not anticipate that Spyglass would limit network bandwidth.

5.2 Host Denial of Service

A serious potential attack that an adversary could launch is to deny service to other clients connecting to the Spyglass. To test this, we spawned a new container and ran a command to fill the container disk (`dd if=/dev/zero of= temp`). This in turn caused the host disk to fill. The system was still able to spawn new containers after the disk was full; however, their auditing processes were quickly killed off as the host ran out of disk space.

One solution is to use the `devicemapper` backend for Docker container storage. Using the `devicemapper` backend allows for finer-grained control on storage by specifying a base size for all container images. However, this means that all containers must be the same size; by default this value is 10 gigabytes. This effect can be mitigated by starting the Docker daemon with the `--storage-opt dm.basesize=1G` option; however, this breaks compatibility with the container creation web interface. Work towards user namespaces and individual quotas will make it trivial to apply file system quotas to containers; however, these features are not yet available.

5.3 Network Protection

The container host is connected to two different networks that are used to provide authentication and audit log storage support for the system. These networks should not be exposed to the user who is looking to access the protected network. By default, the container does not have access to use the `ping` command. However, it was still possible to use `netcat` to send data between two hosts if the destination address was known. It was also possible to connect back out to the untrusted network, which would allow an attacker to pivot to another host.

We implemented firewall rules on the Spyglass container host to mitigate these and other network-based attacks. This way, traffic to and from the container was limited to SSH inbound and a select set of outbound protocols for the container. Finally, the host firewall explicitly drops and logs all connection attempts from the container to the host. In our testing, we found no unauthorized network connections were allowed.

5.4 Container Escalation and Escape

A core assumption of the security of our system is that a user cannot escalate privileges and/or "break out" of the container itself. We accomplish this by proper configuration and multilayered defensive practices.

We configure the container in such a way that a remote user does not have root privileges. This is to protect against an escape attack within system, as it is easier to jump from the container to the container host if an attacker has root inside the container. We also suggest regular rebuilding of the container with patched binaries. This makes it more difficult for an attacker to take advantage of a root exploit in any base packages.

The use of mandatory access control can also limit the scope of an attacker who is able to both escalate to a root user within the container *and* break out of the container itself. Docker has SELinux rules available for use with Red Hat Enterprise Linux 7 and derivatives, but our implementation uses Ubuntu. In future work, we plan to implement this addition protection that would further raise the bar for an adversary trying to compromise Spyglass.

5.5 Audit Security

The logs created by the SudoSH process running inside of the container are ephemeral. To address this issue, we send the logs to another host on a separate network to provide a record in the event of a container compromise or other security event. `rsync` provides functionality to move files over to the audit host. We discuss methods to optimize this approach in Section 7.

6 Related Work

There is a variety of work that show early interest and effort into implementing container-based solutions to insulate a host operating system from attack. Ioannidis et al. implement a tag that is attached to files obtained from remote sources that allows built-in limiting what malicious code can do to a user's other files [19]. This is interesting, in that modern operating systems have implemented a variation of this idea (Apple's Mac OS X is able to detect files that have been downloaded and warn before opening); however, the technology that is more applicable to this project has gone largely unimplemented in major operating systems. Wagner also shows

#	Memory Use	CPU Cycles	Real Time	User Time	System Time
1	4.80	320437210	0.77	0.01	0.02
2	4.80	246014871	0.85	0.02	0.03
3	4.91	464523389	0.16	0.00	0.00
4	4.79	417975143	0.16	0.01	0.00
5	4.80	332404388	1.05	0.01	0.00

Table 1: Memory (*MB*), CPU (*jiffies*) and Time (*seconds*) for container instantiation.

early interest in the idea of containerization, and implements a method of attempting to "containerize" an application in user space by monitoring system calls [33]. Wagner monitors system calls and the files they act upon against a policy to ensure that applications are allowed to access specific files or network devices. The approach comes about a year before the release of SELinux, which uses contexts rather than per-application configuration to enforce access to resources.

Thakwani proposes a new UNIX `dfork()` call that instantiates the child process in a virtualized machine [32]. This solution is elegant in that it provides a very low-level means to ensure that processes start in separate namespaces. Thakwani's work doesn't measure the amount of time it takes to use `dfork()` with a new virtual machine on each use. Many processes can be sandboxed in the same virtual machine in Thakwani's architecture, thus saving time; however, this would not work well for our goal of isolating users from each other.

Parno et al. demonstrates demand-based virtualized containers that are instantiated upon user-login to a website in CLAMP [26]. CLAMP goes on to actively broker access to a particular database and ensure each container instance only contains the appropriate data for the authenticated user. While our work does not deal with specific user data, CLAMP demonstrates a model of mitigating risk by implementing nonpersistence and containerization.

Similarly, Huang et al. propose a framework to reduce an adversary's ability to have an attack persist on a particular network by refreshing to a known clean state on a regular basis [14]. This methodology works well on detectable and undetectable attacks thanks to the regular refresh interval. However, it does not protect against any lower-level (i.e., hardware) attacks that may occur [24]. It also provides some form of "highly available" architecture to handle the hosts that are being actively refreshed. Spyglass makes no guarantee of a highly available resource, but new containers are easy to instantiate unless the container host has failed.

Our approach is similar to the Lightweight Portable Security [22]. Lightweight Portable Security creates a bootable, read-only environment that doesn't store state. This affords an organization reasonable assurance that there is no persistent malware on a machine they may not own, which addresses concerns in Section 2.2 regarding virtual private networks. However, the technique has a significant amount of overhead in that it requires a user to reboot into the environment, and it makes no assumptions about attacks that would live in hardware (and therefore, persist across reboots) [24].

Nonpersistence can have operational benefits as well. An example of this is Ganger, a tool for instantiating containers when a network request is received [31]. The motivation for Ganger was to create a temporary environment that would ensure that files created under `/tmp` would be cleaned up in an orderly fashion after the network connection was closed. This was due to the use of a particular application that wrote a large amount of temporary data.

Proving that there is a market for monitoring of the connection concentrator, Pythian's *Adminiscope* implements a form of connection concentrator to a private network with live auditing ability [28]. However, it is unclear as to what mechanisms are implemented to guard the host against compromise and other threats to the concentrator itself. Similarly, another industry product exists named Invincea [18]. Invincea brings together concepts of non-persistence and isolation to protect a browser against web-based malware. In our system, we aim to protect sensitive infrastructure from a bad client.

A similar commercial offering is Dome9's Secure Access Leasing product [6]. Secure Access Leasing is a mechanism by which users request access to various cloud-hosted resources, and the Dome9 product has an agent that configures hosts and AWS firewalls to allow a particular user access to the host for a certain amount of time. The solution allows administrators to see when users are accessing which resources. This is an easy win for many organizations with assets in the cloud. However, an organization has no visibility into what a particular administrator is doing with that resource; the auditing is pushed off to the host that needs to be accessed.

Recently, Yelp created `dockersh`, a shell environment that is able to provide nonpersistent shell environments for users who SSH into a server [35]. This is one of the closest matches to what the system aims to do. The `dockersh` documentation does mention the issues regarding opening bastion hosts to the Internet. The system described runs a SSH daemon in the container envi-

ronment, which does allow for more separation. There is also limited discussion of good security practice in the event of a compromise. Users blindly implementing `dockersh` against the warnings of the engineers at Yelp will find themselves without any situational awareness in the event of a compromise. We mitigate these concerns by providing a "belt and suspenders" approach to security. If our container is compromised, we do have a log for a period of time that allows us to replay the attacker's movements pre-compromise.

7 Future Work

While we were able to create a system design and architecture that meets our needs and goals, there are several considerations that could enhance the security, usability, and performance of the system. Some of these items include:

- Centralized authentication is prevalent in many organizations, and it may be beneficial to leverage that as an authentication backend for the container request application.
- While it is convenient that the SudoSH utility logs all keystrokes, there are instances where this is a problem (e.g., when a user enters a password). Creating a mechanism to ignore sensitive details would be important to mitigate some insider risk.
- The container audit daemon executes rsync twice for every keystroke. We plan to implement a simple streaming data service on top of an SSH tunnel to a corresponding agent on the audit host to lower overhead for audit information.
- Migration to a Red Hat Enterprise Linux-based system would allow the use of SELinux for greater container security. Later versions of Docker will also support user namespaces, which improves the security of containers to break out even when an adversary can obtain root access inside of the container.
- A means for keeping track of the age of administrator keys, and enforcing age limits on those keys.
- Providing the SSH host key signature to the web interface (so a user could verify the key of the container they are connecting to) would be an important addition to ensure the security of the connection from a man-in-the-middle attack. This is especially relevant in Spyglass as the "trust on first use" nature of SSH host authentication provides limited benefit when the containers are re-instantiated on each session.

8 Conclusion

Given that external attackers and malicious system administrators could wreak havoc across an organization's network, it is extremely important to protect access to networks with sensitive interfaces connected to them. In this paper, we presented Spyglass, a system that utilizes auditability, nonpersistence, isolation, and multi-factor authentication to protect sensitive networks. This system requires minimal change to the actual configuration of the network, provides a high security bastion, and allows an organization to securely audit their administrators' activity.

The container request application, container daemon, and audit daemon are in the process of being open sourced. The project will be updated as necessary, and pull requests from the community are welcome.

9 Acknowledgements

This paper is derived from "Demand-Provisioned Linux Containers for Private Network Access," a project completed in partial fulfillment for the degree of Master of Science in Networking and System Administration at Rochester Institute of Technology [16]. We thank Steve Stonebraker for comments and suggestions on the early design concept. Finally, we thank Thomas Moyer and Stephanie Mosely for their help reviewing this paper.

References

[1] S.M. Bellovin and W.R. Cheswick. Network firewalls. *Communications Magazine, IEEE*, 32(9):50–57, Sept 1994.

[2] Peter Bright. JPMorgan Chase hack due to missing 2-factor authentication on one server. `http://arstechnica.com/security/2014/12/jpmorgan-chase-hack-because-of-missing-2-factor-auth-on-one-server/`, Dec 2014.

[3] Pamela Brown, Jim Sciutto, Evan Perez, Jim Acosta, and Eric Bradner. Investigators think hackers stole Sony passwords. `http://www.cnn.com/2014/12/18/politics/u-s-will-respond-to-north-korea-hack/index.html?sr=tw121814nksysadminsony620pVODtopLink`, Dec 2014.

[4] Paul C. Brutch, Tasneem G. Brutch, and Udo Pooch. Indicators of UNIX Host Compromise. *;login:*, Sep 1999. `https://www.usenix.org/legacy/publications/login/1999-9/features/compromise.html`.

[5] Australian Signals Directorate. Strategies to Mitigate Targeted Cyber Intrusions. `http://www.asd.gov.au/publications/Mitigation_Strategies_2014.pdf`, Feb 2014.

[6] Dome9. Secure Access Leasing. http://www.dome9.com/overview/secure-access-leasing.

[7] Dan Farmer. Sold Down the River. http://fish2.com/ipmi/river.pdf, Jun 2014.

[8] Harriet G. Goldman. Building Secure, Resilient Architectures for Cyber Mission Assurance. Technical report, MITRE, 2010.

[9] Dan Goodin. 25-gpu cluster cracks every standard windows password in 6 hours. http://arstechnica.com/security/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/, Dec 2012.

[10] Dan Goodin. Aws console breach leads to demise of service with "proven" backup plan. http://arstechnica.com/security/2014/06/aws-console-breach-leads-to-demise-of-service-with-proven-backup-plan/, Jun 2014.

[11] Google. cAdvisor. https://github.com/google/cadvisor.

[12] Douglas Hanks. SudoSH. http://sourceforge.net/projects/sudosh/, 2013.

[13] Ming-Yuh Huang. Critical Information Assurance Challenges for Modern Large-Scale Infrastructures. In Vladimir Gorodetsky, Igor Kotenko, and Victor Skormin, editors, *Computer Network Security*, volume 3685 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin Heidelberg, 2005.

[14] Y. Huang, D. Arsenault, and A Sood. Incorruptible system self-cleansing for intrusion tolerance. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 4 pp.–496, April 2006.

[15] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.

[16] Patrick T. Cable II. Demand-Provisioned Linux Containers for Private Network Access. Master's thesis, Rochester Institute of Technology, Dec 2014.

[17] InfluxDB. http://influxdb.com.

[18] Invincea. Freespace. http://www.invincea.com/how-it-works/containment/.

[19] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating Systems: A New Approach to Application Security. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 108–115, New York, NY, US, 2002. ACM.

[20] Loic Jaquemet. SSLSnoop. https://github.com/trolldbois/sslsnoop.

[21] P.G. Kelley, S. Komanduri, M.L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 523–537, May 2012.

[22] Lightweight Portable Security. http://www.spi.dod.mil/lipose.htm.

[23] Robert Love. Kernel Korner: Intro to Inotify. *Linux J.*, 2005(139):8–, November 2005.

[24] H. Okhravi, M.A. Rabe, W.G. Leonard, T.R. Hobson, D. Bigelow, and W.W. Streilein. Survey of Cyber Moving Targets. Technical report, MIT Lincoln Laboratory, Jul 2013.

[25] OpenVPN. http://www.openvpn.net.

[26] B. Parno, J.M. McCune, D. Wendlandt, D.G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 154–169, May 2009.

[27] Mike Pope. Securely connect to linux instances running in a private amazon vpc. http://blogs.aws.amazon.com/security/post/Tx3N8GFK85UN1G6/Securely-connect-to-Linux-instances-running-in-a-private-Amazon-VPC, Sept 2014.

[28] Pythian. Adminiscope. http://www.pythian.com/products/adminiscope/.

[29] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems*, volume 8788 of *Lecture Notes in Computer Science*, pages 77–93. Springer International Publishing, 2014.

[30] SANS Institute. Critical Security Controls. http://www.sans.org/critical-security-controls.

[31] Andy Sykes. Ganger. https://github.com/forward3d/ganger.

[32] Ashish Thakwani. Process-level Isolation using Virtualization. Master's thesis, North Carolina State University, Jan 2010.

[33] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Master's thesis, University of California, Berkeley, 1999.

[34] Zachary Wikholm. CARISIRT: Yet Another BMC Vulnerability (And some added extras). http://blog.cari.net/carisirt-yet-another-bmc-vulnerability-and-some-added-extras/, 2014.

[35] Yelp. HACK209 - dockersh. http://engineeringblog.yelp.com/2014/08/hack209-dockersh.html.

[36] Yubico. Yubikey standard. http://www.yubico.com/products/yubikey-hardware/yubikey/.

[37] Elizabeth Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. O'Reilly Media, second edition, 2000.

DevOps is Improv: How Improv Made Me a Better Sysadmin

Brian Sebby, *Argonne National Laboratory*

Abstract

With the rise of DevOps as a prevailing software development model, organizations are finding that teamwork and communication are tools that are vital to the overall success of their mission. Three years ago, I took my first improv class, and in addition to helping me be more comfortable with public speaking, I have found that many of the techniques that help an improv team succeed on stage are directly applicable to helping a DevOps team succeed.

1. Saying "Yes And"

While there are many forms of improvisation, all of them share one major rule: saying "Yes And". Saying "Yes And" means that you are agreeing with the reality that is being established on stage and then adding more to it. In a DevOps team, saying "Yes And" means that you are open to new ideas, and are adding ideas of your own, rather than denying the contributions of your teammates. Saying "Yes And" also requires you to listen carefully to what is being said so that your contributions can add to the whole performance or project.

2. Supporting Your Team

Almost all forms of improvisation are performed by teams, and for a performance to be a success, the team must work together and support each other. It's common before a performance for improvisers to tell each other "I've got your back", and that is also needed in DevOps. If there is a problem or someone needs help, a team should work together to solve it, rather than try to find a way to assign blame. On stage, providing support to your scene partners makes them look good, and it ensures that when you are in the spotlight, your team will be there to provide the support you need to look good.

3. Taking Risks

Improv provides a heightened view of reality, and in order to provide humor for the audience, improvisers must take risks and take advantage of slips of the tongue and other unexpected statements or developments in the scene. Rather than being discouraged, these are called "gifts", and can take the performance in directions that no one expected. In DevOps, if we believe that our team has our back, it is much easier to take risks - and we may find that unexpected results may improve a project and make it more successful than it may have been otherwise.

4. Shared Success

On any team, whether in improv or in DevOps, individual members of the team will have various skill levels. At the end of a performance, however, the audience is not applauding for one player - they are applauding for the team. If a stronger player provides support for the other members of a team, that will help to ensure that the show is a success. Likewise, in DevOps, success or failure is shared between the whole team, and completing a project successfully relies on the team working together.