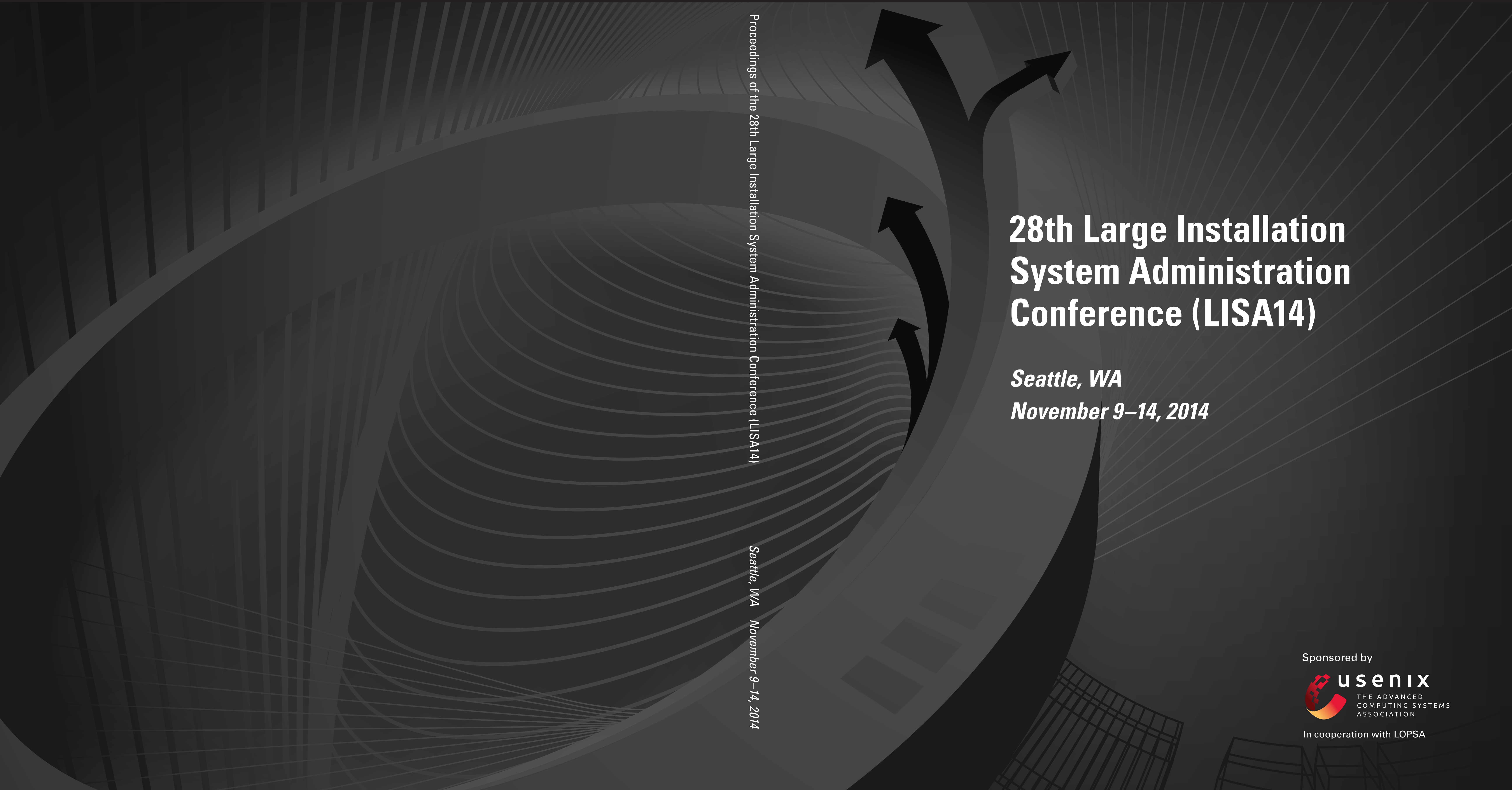


usenix
ASSOCIATION

Proceedings of the 28th Large Installation System Administration Conference (LISA14)

*Seattle, WA November 9–14, 2014*

conference

*proceedings*

# 28th Large Installation System Administration Conference (LISA14)

***Seattle, WA***

***November 9–14, 2014***

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with LOPSA

## Thanks to Our LISA14 Sponsors

### Gold Sponsors

CAMBRIDGE Computer ARTISTS IN DATA STORAGE

Google

ORACLE

### Silver Sponsors

CITRIX Open@Citrix

DreamHost

openstack CLOUD SOFTWARE

puppet labs

redhat.

### Bronze Sponsors

Center for Internet Security

cloudera

Pythian love your data

vmware

### Media Sponsors and Industry Partners

ACM *Queue*
*ADMIN*
Cascadia IT Conference
CRC Press
DevOps.com
Distributed Management Task Force (DMTF)
EnterpriseTech
FreeBSD Foundation
HPCwire
InfoSec News
KDE
*Linux Pro Magazine*
LinuxCon Europe
LinuxFest NorthWest
LXer
No Starch Press
O'Reilly Media
*Raspberry Pi Geek*
SNIA
ThinkGeek
UserFriendly.Org

ISBN 978-1-931971-17-1

## Thanks to Our USENIX and LISA SIG Supporters

### USENIX Patrons

Google  Microsoft Research  NetApp  VMware

### USENIX Benefactors

Akamai  Facebook  Hewlett-Packard
IBM Research  *Linux Pro Magazine*  Puppet Labs

### USENIX and LISA SIG Partners

Cambridge Computer  Google  Can Stock Photo

### USENIX Partners

EMC  Huawei

USENIX Association

# Proceedings of the 28th Large Installation System Administration Conference (LISA14)

November 9–14, 2014
Seattle, WA

## Conference Organizers

## External Reviewers

| | | |
|---|---|---|
| Paul Armstrong | Matt Disney | Adam Moskowitz |
| Yanpei Chen | Suzanne McIntosh | Josh Simon |
| Jennifer Davis | Dinah McNutt | Guanying Wu |

# Proceedings of the 28th Large Installation System Administration Conference (LISA14)

## Message from the LISA14 Program Chair

Welcome to LISA14! This marks the 28th meeting of the USENIX Large Installation System Administration Conference, and I'm pleased to present this year's program and proceedings.

This year, we shook things up a bit, pulling together all aspects of the LISA conference under coordination of the Chair. This resulted in a carefully curated program with content that spans tutorials, workshops, and technical talks, providing attendees with the opportunity to tailor the conference to their needs. We're excited about the coordinated content and the tracks, and we think you will be, too.

Of course, I could not have done this without a fantastic team of conference organizers who coordinated content across several areas to deliver the amazing program you see here. I would also like to thank those who contributed directly to the program: authors, shepherds, external reviewers, speakers, tutorial instructors, attendees, the LISA Build team, USENIX Board of Directors liaisons, and the USENIX staff. I would also like to personally thank USENIX Executive Director Casey Henderson for her support as we chartered new territory in conference organization, and for securing Diet Coke for the conference venue. Casey has been wonderful to work with and a true teammate and colleague.

This year, the conference accepted nine peer-reviewed full research papers, resulting in an acceptance rate of 31%. The conference also accepted nine peer-reviewed posters, which appear in the conference proceedings as extended abstracts. This adds to USENIX's considerable body of peer-reviewed published work. To foster interaction and discussion among the LISA community and to extend the reach and engagement of our research papers, all of our accepted papers have been invited to present a poster in addition to their paper presentation. We hope our attendees will take the opportunity to learn about the cutting-edge research that is happening in systems administration, and find out how it might affect their work in the months and years to come.

I am very proud of this year's program and hope you enjoy the conference. Most importantly, I hope you also enjoy the hallway track; say "hi" to old friends and meet some new ones. Don't be afraid to introduce yourself to someone new! The strength of LISA lies in you, the attendees, and in the discussions and technical conversations that happen between sessions.

**Nicole Forsgren Velasquez,** ***Utah State University***
**LISA14 Program Chair**

# HotRestore: A Fast Restore System for Virtual Machine Cluster

Lei Cui, Jianxin Li, Tianyu Wo, Bo Li, Renyu Yang, Yingjie Cao, Jinpeng Huai
*State Key Laboratory of Software Development Environment*
*Beihang University, China*
{*cuilei, lijx, woty, libo, yangry, caoyj*}*@act.buaa.edu.cn* {*huaijp*}*@buaa.edu.cn*

## Abstract

A common way for virtual machine cluster (VMC) to tolerate failures is to create distributed snapshot and then restore from the snapshot upon failure. However, restoring the whole VMC suffers from long restore latency due to large snapshot files. Besides, different latencies would lead to discrepancies in start time among the virtual machines. The prior started virtual machine (VM) thus cannot communicate with the VM that is still restoring, consequently leading to the TCP backoff problem.

In this paper, we present a novel restore approach called HotRestore, which restores the VMC rapidly without compromising performance. Firstly, HotRestore restores a single VM through an elastic working set which prefetches the working set in a scalable window size, thereby reducing the restore latency. Second, HotRestore constructs the communication-induced restore dependency graph, and then schedules the restore line to mitigate the TCP backoff problem. Lastly, a restore protocol is proposed to minimize the backoff duration. In addition, a prototype has been implemented on QEMU/KVM. The experimental results demonstrate that HotRestore can restore the VMC within a few seconds whilst reducing the TCP backoff duration to merely dozens of milliseconds.

## 1 Introduction

Machine virtualization is now widely used in datacenters and this has led to lots of changes to distributed applications within virtualized environments. In particular, the distributed applications are now encapsulated into virtual machine cluster (VMC) which provides an isolated and scaled computing paradigm [1, 24, 22]. However, failures increasingly become the norm rather than the exception in large scale data centers [17, 35]. The variety of unpredictable failures might cause VM crash or network interruption, and further lead to the unavailability of applications running inside the VMC. There are many approaches for reliability enhancement in virtualized environment. Snapshot/restore [25, 39, 40, 37] is the most widely used one among them. It saves the running state of the applications periodically during the failure-free execution. Upon a failure, the system can restore the computation from a recorded intermediate state rather than the initial state, thereby significantly reducing the amount of lost computation. This feature enables the system administrators to recover the system and immediately regain the full capacity in the face of failures.

In the past decades, several methods have been proposed to create distributed snapshot of VMC, and most of them aim to guarantee the global consistency whilst reducing the overhead such as downtime, snapshot size, duration, etc [11, 25, 14]. However, restoring the VMC has received less attention probably because restoration is only required upon failures. As a matter of fact, due to the frequently occurring failures in large scale data centers, restoration becomes frequent events accordingly [35]. Worse still, just one VM crash would lead to the entire VMC's restoration with the consistency guarantee taken into account. The frequent restoration of multiple virtual machines cause non-negligible overheads such as restore latency and performance penalty. Here, restore latency is referred to the time to load saved states from the persistent storage until the VM execution is resumed.

There are a few well-studied works on improving VM restoration. *Working set restore* [39, 40] is one solution proposed recently, and it restores a single VM by prefetching the working set. Since the working set reflects the access locality, this method reduces the restore latency without compromising performance. It seems that the VMC restore could be simply accomplished by restoring the VMs individually with *working set restore* [19]. Unfortunately, several practical drawbacks are still far from settled.

First, the restore latency with *working set restore* is still long. In fact, the latency is proportional to the work-

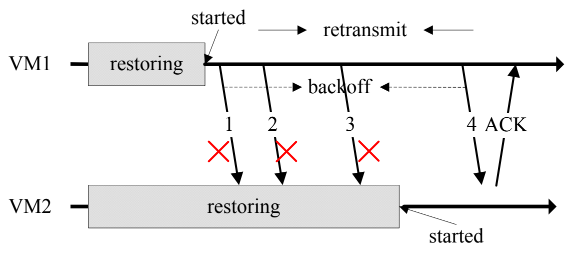


Figure 1: A TCP transmission case during restoration. VM1 after start sends packet1 to VM2, and this packet will be lost because VM2 which is restoring is currently suspended. VM1 would resend this packet once retransmission timeout (RTO) is reached. Packets 2, 3, 4 are retransmitted items of packet1. If packet1 is a SYN to connect VM2, the TCP handshake would fail if the retransmission duration exceeds TCP connection timeout. If packet1 is a heartbeat packet, the timeout may cause VM2 be falsely reported as fail which would result in misbehaviour.

ing set size which is directly related to the workload activity. For memory intensive tasks, the working set might be the entire memory, so that the *working set restore* degrades to *eager restore*[37] which starts the VM after all state is loaded.

Second, due to the heterogeneity of virtual machines cooperated in the VMC as well as the variety of workloads, the working set sizes of various VMs may be different. This difference results in diverse restore latencies and hence causes VMs being launched at different times. As a result, if the prior started VM sends a packet to a restoring one, it will not receive any reply within a while. This might lead to temporary backoff of active TCP connections as illustrated in Figure 1.

Therefore, in the cluster, the practical VM disruption time not only involves the restore latency but is determined by the TCP backoff duration as well. The backoff duration directly depends on the degree of discrepancy among VMs' restore completion times, i.e., start times. What's more, due to the complexity of workloads, these VMs may differ greatly in working set sizes, making TCP backoff duration the dominant factor in disruption.

In this paper, we propose HotRestore, which is capable of restoring the saved state of VMC swiftly and efficiently. Unlike prior VMC restore tools, HotRestore could resume the applications' execution within a few seconds and the applications can regain their full capacity rapidly. Moreover, the distributed applications only suffer transient network interruption during the restoration. Consequently, HotRestore can be naturally adopted in scenarios with high availability requirement, where fast recovery is essentially critical to provide reliable services to end users.

HotRestore proposes two key ideas to achieve these objectives. On one hand, it traces the memory access during *post-snapshot*, and records the traced pages in a first-access-first-load (FAFL) queue, which finally constitutes an elastic working set. The motivation behind is that when a VM is restored, it will be roll-backed to the snapshot point. If the execution is deterministic, the VM will re-execute in the same way as that of *post-snapshot*. As a result, the traced memory pages during *post-snapshot* will be touched again and thus can be regarded as working set pages. By only loading the working set pages upon restoration, the restore latency decreases with a lot.

On the other hand, restore line which depicts the start times of VMs is designed in order to reduce the TCP backoff duration. The basic idea is that for a packet sent from one VM, the associated destination VM must have been started to receive the packet for preventing potential backoff. The restore line derives from defacto restore order and causal restore order. The former one is revealed by the calculated working set sizes of VMs while the latter is communication-induced. Since the semantics of the two orders might conflict, HotRestore revises the working set sizes to make defacto order be consistent with causal order, and thereafter computes the restore line. Moreover, a restore protocol is designed to guarantee that the VMs can start as indicated by the restore line, thereby significantly minimizing the backoff duration.

Our contribution is three-fold. First, we introduce elastic working set, which is a subset of active memory pages in any desired size, in order to restore a single VM rapidly. Second, we propose restore line for virtual machines that cooperate into a cluster, schedule the VMs' start times to minimize the TCP backoff duration. Third, we have implemented HotRestore on our previous work called HotSnap [14] which creates distributed snapshots of the VMC, and conducted several experiments to justify its effectiveness.

The rest of the paper is organized as follows. The next section gives a brief overview of previous work on HotSnap. Section 3 presents the concepts and implementation of elastic working set. Section 4 introduces the algorithm to compute the restore line and describes the restore protocol. Section 5 introduces several implementation details on QEMU/KVM platform followed by the experimental results in Section 6. Finally we present the previous work related to HotRestore in section 7 and conclude our work in Section 8 and 9.

## 2 A Brief Overview of HotSnap

HotSnap creates a global consistent state among the VMs' snapshots. It proposes a two stage VM snapshot creation approach consisting of *transient snapshot* and *full snapshot*. In *transient snapshot*, HotSnap suspends the VM, records the CPU and devices' state, sets guest memory pages to be write-protected, and then starts the VM. After that, *full snapshot* starts up. HotSnap will save the guest pages in a copy-on-write manner. Specifically, upon a page fault triggered by touching the write-protected page, HotSnap will save the page into snapshot file, remove the write-protect flag and then resume the VM. The recorded snapshot is actually the instantaneous state in *transient snapshot*; therefore *full snapshot* is actually a part of *post-snapshot* stage.

In HotSnap, we tailor the classical message coloring method to suit to virtualized platforms. In the coloring method, the packet color is divided into *pre-snapshot* and *post-snapshot*, so is the VM color. HotSnap intercepts the packet sent and received by the simulated tap device on QEMU/KVM. For a packet to be sent, HotSnap piggybacks the packet with the immediate VM color. For a received packet, if the packet color is *post-snapshot* while the VM color is *pre-snapshot*, the packet will be temporarily dropped to avoid inconsistency; otherwise, the packet will be forwarded to the virtual machine and be finally handled.

The consistency of global state is guaranteed during snapshot creation by HotSnap. Therefore, HotRestore makes no attempt to ensure consistency upon restoration, which is different to previous works that focus on consistency as well as avoiding domino effect [16, 34, 36].

## 3 Elastic Working Set

This section first presents the estimation method of elastic working set, and then describes how to create snapshot and restore the VM when working set is employed.

## 3.1 Working Set Estimation

An elastic working set should satisfy three requirements. Firstly, the restore latency is as short as possible without comprising application performance. Secondly, high hit rate and high accuracy are achieved after VM starts. High hit rate implies most of the accessed pages hit in the working set, while high accuracy means most of the working set pages will be touched within a short while after VM starts. Thirdly, the working set size could scale up or scale down without the decrement of hit rate, accuracy and performance. In this manner, the size can be revised on-demand upon VMC restoration. To estimate one such working set, we need to determine: i) which pages should be filled into the working set, and ii) how many pages should be loaded on restoration (or namely the working set size).

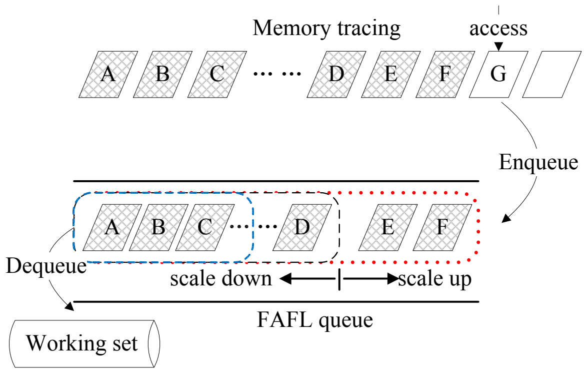


Figure 2: Elastic working set with the FAFL queue.

### 3.1.1 Working Set Pages

Upon restoration, the VM will be roll-backed to the saved snapshot point and continue to execute. Ideally, if the execution is deterministic, the VM after restoration will re-execute the same instructions and touch the same memory area as that of *post-snapshot*. This insight inspires us to trace these memory access during *post-snapshot* and regard the traced pages as candidate working set pages for producing the final working set.

Optimal selection from the candidate pages has been well studied. LRU and CLOCK are two classical and widely-used page replacement algorithms [23, 41]. However, we argue that the FIFO manner could be better in the mentioned scenario within the paper. The reason is as follows. If the execution is deterministic, the page access order after restoration remains the same as that of *post-snapshot*. This implies that the first accessed page during *post-snapshot* will be firstly accessed after restoration. As a result, HotRestore adopts a first-access-first-load (FAFL) manner to determine the working set pages. Specifically, HotRestore saves the traced pages in the FAFL queue during *post-snapshot*, and loads the pages dequeued from the queue upon restoration. Figure 2 illustrates the overview of the FAFL queue. The traced pages, i.e., A-F, have been stored in the queue in access order, and the working set can be simply produced by dequeuing the pages from the queue.

Elasticity is another crucial feature of the FAFL queue. The queue depicts the order of pages which are touched by guest kernel or applications during *post-snapshot*. On tone hand, given that the VM execution is deterministic, loading more or fewer pages into the working set upon restoration will not influence the hit rate or accuracy seriously. Consequently, it enables the working set to scale up/down efficiently. On the other hand, the background load thread after VM is launched could still fetch the pages in the FAFL manner rather than loading the pages blindly. In this way, the probability of page fault could be effectively reduced. This is essentially important to maintain the application performance without severe degradation when the working set size decreases.

The external inputs events or time change make the VM execution be non-deterministic invariably in real world scenarios. Despite this, our experimental results[1] show that elastic working set could achieve high hit rate and accuracy. Meanwhile, it could scale up and scale down without compromising performance.

### 3.1.2 Working Set Size

The working set size plays an important role when rolling back. Although a small size reduces the restore latency, it incurs numerous page faults, and thus aggravating the decreased performance after restoration. On the other hand, a large size could avoid severe performance penalty incurred by more loaded memory pages, but at the cost of long latency.

Statistical sampling approach described in ESX Server [38] is frequently used to predict the working set size. However, it lacks the ability to respond to the phase changes of workload in real time which is critical in restoration. Since the working set size upon restoration should capture the size of workload activity in a timely manner, we propose a hybrid working set size estimation method. First, we adopt a statistical sampling approach to calculate the working set size during normal execution. The size depicts the average workload activity in a long period, and is referred to as $WSS_{sample}$. Secondly, we count the touched pages during *post-snapshot*. The page count reflects timely workload activity in part, and is referred to as $WSS_{snapshot}$. The expected working set size is the weighted sum of these two sizes and is determined by: $WSS = \alpha * WSS_{sample} + \beta * WSS_{snapshot}$

In most programs, $WSS$ remains nearly constant within a phase and then changes alternately. Since the workload always keeps steady for quite a while [41], we empirically set $\alpha$ to be larger in HotRestore, i.e., $\alpha$ is 0.7 and $\beta$ is 0.3. In addition, due to the well scalability of the FAFL queue, we adopt $WSS/2$ rather than $WSS$ as the actual working set size upon restoration. The remaining pages will be loaded by the background thread from the FAFL queue or by demand-paging. The experimental results in §6.1.3 demonstrate that the size shrink only incurs negligible performance overhead.

## 3.2 Snapshot and Restore

This section describes the VM snapshot and restore when elastic working set is employed.

**Snapshot.** Unlike HotSnap which only traces the memory write operations, HotRestore traces all accesses and records them in the FAFL queue to estimate the working set. Therefore, HotRestore adopts a snapshot approach that consists of copy-on-write and record-on-access. In detail, HotRestore sets the guest memory pages to be non-present to trace all accesses. For write operation, HotRestore records the page frame number into the FAFL queue, saves the page content into persistent storage, and then removes the non-present flag to allow the page to be read or written without fault later on. For read operation, HotRestore only records the page frame number but does not save the page content. This is because read operations occur much more frequently than write operations, and saving the page content would lead to serious performance degradation. Therefore, HotRestore removes the read-protect flag but reserves the write-protect flag. The page content will be saved either in copy-on-write manner once it is required to be written or by the background copy thread.

**Restore.** Upon restoration, HotRestore firstly loads the CPU state and devices' state, and then fetches the working set pages into the guest memory. Afterwards, it sets the page table entries of unrestored pages to be non-present before starting the VM. The unrestored pages will be loaded by i) on-demand paging due to touching the non-present page after VM starts, or ii) background load thread running concurrently which ensures that restore finishes in a reasonable period of time.

## 4 Restore of Virtual Machine Cluster

The key of VMC restore is to mitigate the TCP backoff problem. In TCP, after sending a packet, the sender will wait for the *ack* from the receiver. It would resend the packet once timeout occurs to ensure that the packet is successfully received. Motivated by this, the basic idea to avoid TCP backoff is to ensure the receiver start before the sender.

This section presents our solution to VMC restore. We start by describing the communication-induced restore dependency graph (RDG). Based on RDG, we compute the causal restore order[2] of VMs, and then schedule the restore line by revising the working set sizes of VMs. Finally, we introduce the restore protocol which ensures the VMs start as indicated by the restore line.

## 4.1 Restore Dependency Graph

We define "$VM_i$ depends on $VM_j$" or $VM_i \rightarrow VM_j$ if $VM_i$ sends a packet to $VM_j$. If $VM_i$ sends a packet to $VM_j$ during snapshot, it will resend the packet after it is restored to the saved snapshot point. Therefore, the dependency

[1]§6.1 will explain the results.

[2]The restore order here describes the order of completion times of the working set restore, or namely VM start times, rather than restore start times.

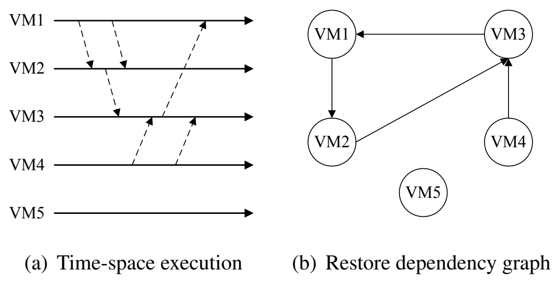


Figure 3: Communication-induced RDG.

can be reserved upon restoration (we guarantee the dependency by deterministic communication in §5.1). This motivates us to construct the RDG via the dependency among VMs. In RDG, each node represents one VM, and a directed edge is drawn from $VM_i$ to $VM_j$ if $VM_i$ depends on $VM_j$. The name "restore dependency graph" comes from the observation that if there is an edge from $VM_i$ to $VM_j$ and $VM_i$ is to be restored, then $VM_j$ must be restored no later than $VM_i$ to be ready to receive the packet and make the reply. Restore here means the "restore end" or "VM start", rather than "restore start". The dependency in RDG is transitive, i.e., if $VM_i \rightarrow VM_j$ and $VM_j \rightarrow VM_k$, then $VM_i \rightarrow VM_k$.

Figure 3(b) demonstrates a RDG yielded by the time-space diagram in Figure 3(a). RDG only depicts the dependency between different VMs, i.e., space, but has no concept of time whether it is physical time or global virtual time. This is because the packet order and send/receive time may change after restoration due to non-deterministic execution. The lack of time semantics allows the existence of dependency ring, e.g., VM1, VM2 and VM3 in Figure 3(b) form a ring. We define $VM_i \leftrightarrow VM_j$ if $VM_i$ and $VM_j$ are in a ring. VM5 is an orphan node; it neither depends on any VM, nor is depended by other VMs. The orphan node reflects the case that there is no packet sent or received associated with the node.

## 4.2 Restore Line Calculation

Restore line depicts the desired start times of VMs while guaranteeing the causal restore order of VMs. The causal restore order can be calculated from the restore dependency graph: if $VM_i \rightarrow VM_j$ in RDG, then $VM_j$ should be ahead of $VM_i$ in causal restore order. Moreover, the VM start time (or restore latency) is related to the working set size. The different working set sizes of VMs form the defacto restore order. This motivates us to compute the restore line by the causal restore order as well as the defacto restore order.

### 4.2.1 Causal Restore Order

The causal restore order is the semantics of logical start time; each dependency edge represents one elapsed clock. It is insufficient to obtain a unique causal restore order solely on RDG. First, the RDG is not necessarily a complete graph, e.g., there is no path between VM2 and VM4, implying VM2 and VM4 can be restored independently after VM3. Second, the existence of the dependency ring and orphan node make a lot of choices on the order. As a result, rather than compute a unique order, we instead give the following rules to construct one feasible causal restore order through RDG.

**Rule 1.** The VM, which doesn't depend on other VMs but is depended, is prioritized in causal restore order.

**Rule 2.** $VM_i$ should restore after $VM_j$, if $VM_i$ depends on $VM_j$ while $VM_j$ does not depend on $VM_i$.

**Rule 3.** The VM that depends on several VMs will not restore until all the depended VMs are restored.

**Rule 4.** The VMs can restore independently if no dependency exists between them.

**Rule 5.** The VMs in the ring restore at the same time, i.e., they are placed together in causal restore order.

**Rule 6.** An orphan node may execute independently and therefore will be free in causal restore order.

Rule 1 seeks the VMs that should be restored first. Once found, the VMs that depend on prior restored VMs will join in the causal restore order by Rules 2 and 3. Rule 4 implies that the VMs can restore independently if there is neither direct nor transitive dependency between them. Based on these rules, we can reach two conclusions. First, after the completion of loading the working set, the VM can start if it satisfies any one of Rules 1, 4, 5, and 6. Second, after one VM is started, the VM can start accordingly if it depends on this started VM and satisfies Rule 2 or Rule 3 as well.

### 4.2.2 Defacto Restore Order

The previously calculated working set sizes of VMs are always different due to the phase changes of workload, varieties of workloads or VM heterogeneity. These different sizes lead to different restore latencies and further form the defacto restore order which can be regarded as the semantics of physical start time.

The problem here is that the VM that is prior in causal restore order may have a larger working set size and thus starts later than the latter VM, making the defacto restore order inconsistent with the causal restore order. The inconsistency results in the TCP backoff problem as mentioned above. As a result, revising the working set sizes to match the two semantics becomes one crux of the problem in restoring the VMC. First, the defacto restore order after revision should be consistent with the causal restore order, to avoid TCP backoff. Second, the revised

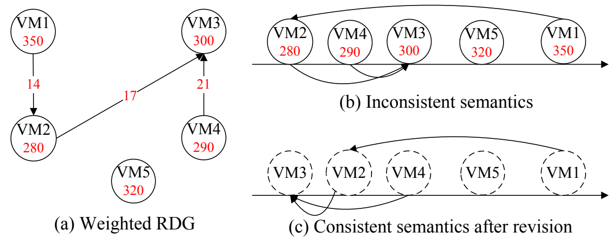

VM1 350
VM2 280
VM3 300
VM4 290
VM5 320
14
17
21
(a) Weighted RDG
(b) Inconsistent semantics
(c) Consistent semantics after revision

Figure 4: The revision of working set sizes. (a) is derived from Figure 3, while the dependency between VM1 and VM3 is removed.

working set size should not impose significant effects on latency or performance for a single VM, i.e., minimum change on previous working set size.

### 4.2.3 Working Set Size Revision

The basic idea of revision is simple. Given that a case that $VM_i \rightarrow VM_j$, but the working set size of $VM_j$, i.e., $S_j$, is larger than $S_i$, we can decrease $S_j$ or increase $S_i$, or change the two both to achieve the matching.

**System Model.** We consider network intensive applications which is common in nowadays large scale data centers, such as distributed database, scientific computing, web services [1], etc. Their characteristic lies in that the communicating VMs send and receive packets frequently. We transfer the RDG to weighted RDG with the assignment to node value and edge weight, as shown in Figure 4(a).

In weighted RDG, the node value $S_i$ is referred to as the previous calculated working set size ($WSS$) of $VM_i$. The edge weight $W_{i,j}$ denotes the number of the captured packets sent from $VM_i$ to $VM_j$ during snapshot. Here, $W_{i,j}$ has no practical meaning, it is just used to ensure that the revised $S_i$, i.e., $S_i^*$, be larger than $S_j^*$ if $VM_i$ depends on $VM_j$. We denote this relation by $S_i^* \geqslant S_j^* + W_{i,j}$. Moreover, this inequation implies that if $VM_i \rightarrow VM_j$ and $VM_j \rightarrow VM_k$, then $S_i^* \geqslant S_k^* + W_{i,j} + W_{j,k}$. $W_{i,j}$ is minor compared to $S_i^*$ or $S_j^*$, hence it causes no significant effect on the revised size. The dependency ring is particular, because the virtual machines in the ring should be provided with equivalent size to start at the same time. Therefore, for $VM_i \leftrightarrow VM_j$, we determine the sizes by:$S_i^* - S_j^* = 0$.

Figure 4(b) demonstrates an inconsistent graph yielded by the weighted RDG in Figure 4(a). The horizontal line shows the defacto restore order, and the arrow depicts the causal restore order. The defacto restore order is [VM2, VM4, VM3, VM5, VM1], while there exist several candidates for causal restore order, e.g., [VM3, VM4, VM2, VM1, VM5], or [VM3, VM2, VM4, VM5, VM1]. Our goal is to reorder the nodes in the horizontal line with the least movement, to guarantee consistency between the causal order and the revised defacto order, such as the example shown in Figure 4(c).

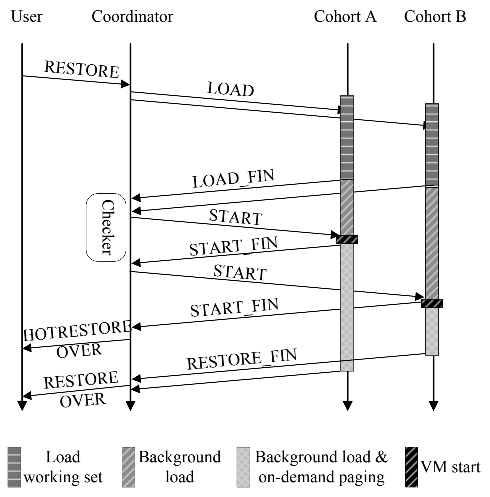


Figure 5: Restore protocol. We assume that B depends on A, so that B starts later.

**Problem Formulation.** We assume that the previous calculated working set size is optimal. This is reasonable since the previous size achieves well tradeoff between restore latency and performance loss for a single VM, as demonstrated in the experimental results in §6.1.3. Given $n$ VMs, let $S = \{S_1, S_2, ..., S_n\}$ be the previous working set sizes of VMs and $W = \{W_{i,j} | VM_i \rightarrow VM_j\}$ be the set of edge weight of each two communicating VMs. We aim to find a minimum revised working set size $S^*$ to guarantee the consistency between the revised defacto restore order and causal restore order. The formulation is denoted by:

$$\begin{array}{llll} \min & \sum_{i=0}^{n} \mid S_i^* - S_i \mid & & \\ s.t. & S_i^* - S_j^* & \geqslant W_{i,j} & (W_{i,j} \in W) \\ & S_i^* - S_j^* & = 0 & (VM_i \leftrightarrow VM_j) \end{array}$$

We use the classical Linear Programming approach [26] to solve this optimization problem. Consequently, the desired restore line is computed through arranging the VMs in ascending order of the revised working set sizes.

## 4.3 Restore Protocol

The purpose of restore protocol is to ensure that the virtual machines start as indicated by the restore line. The restore protocol is entirely software based and does not depend on any specialized hardware. We make the following assumptions about the distributed VMC system. First, the VMs, the underlying physical servers as well as the network devices are failure-free during restoring. This is reasonable since the restoration procedure lasts for a short time compared to the mean time to failure (MTTF) of hardware or software. Second, the network latency is minor, i.e., the messages can be received within a relatively short period. Otherwise, the restore latency would be directly related to the network latency on waiting for the messages of protocol. This assumption is satisfied in nowadays data center networks which always utilize 1Gbps or even 10Gbps Ethernet.

There exist two roles in restore protocol, Coordinator and Cohort, as shown in Figure 5. Each virtual machine is associated with one Cohort, and only one VM is regarded as the Coordinator. The roles of Coordinator consist of constructing the restore dependency graph, scheduling the restore line, broadcasting the working set sizes of VMs and notifying Cohorts to load the working set or start to execute. Besides, the Coordinator employs Checker to determine whether the associated VM can start after receiving the LOAD_FIN reply from the Cohort, and which VMs can start correspondingly once receiving the START_FIN reply from one Cohort, as described in §4.2.1. The Cohort restores the VM through three steps: i) load the working set pages after receiving the LOAD command and then reply LOAD_FIN, ii) load the pages in background until receiving the START command, and iii) start the VM after receiving the START command, reply the START_FIN command, and then load the remaining pages through on-demand paging along with background loading.

Disruption will disappear after all the VMs are started, meanwhile the hot restoration is completed. The whole restoration procedure will not finish until all the VMs reply RESTORE_FIN after completion of loading the remaining memory pages from the snapshot file.

## 5 Implementation Issues

HotRestore is implemented on qemu-kvm-0.12.5 [28] in Linux kernel 2.6.32.5-amd64, it does not require the modification of guest OS. HotRestore utilizes HotSnap to guarantee global consistency of snapshot state. Some optimizations such as compression of zero pages are provided by HotSnap and are also employed in HotRestore. Since the page saved in background belongs to the working set if it is accessed during snapshot procedure, HotRestore stores the working set pages in the snapshot file instead of a separate file. Besides, it creates a FAFL file to store the guest frame number for indexing these pages. The FAFL file makes it convenient to fetch working set pages in any desired size. The rest of this section describes the sub-level parts and optimizations in detail.

### 5.1 Packets Used to Construct RDG

In reliable communication, the receiver will reply *ack* to the sender. The capture of *data/ack* packets makes sender and receiver depend on each other and further form a ring in RDG. The dependency from receiver to sender is compelled, therefore it should be removed when constructing the RDG. One possible approach is to identify the roles by analyzing the packets, however it is extremely complicated and is impractical. Therefore, we resort to the on-the-fly packets, which cannot be received by the destination VM during *transient snapshot*. It cannot be received due to two reasons: i) the receiver VM is suspended to create *transient snapshot* or ii) the packet violates the global consistency, i.e., it is sent from a *post-snapshot* VM to a *pre-snapshot* VM. HotRestore logs on-the-fly packets on the receiver side, and constructs the RDG through these packets. Logging these packets brings two benefits. First, the on-the-fly packet will not be received or handled, and hence avoiding two-way dependency for one transmission. Second, replaying these packets after restoration can ensure deterministic communication, and hence the dependency is preserved.

UDP packets are also preserved. Although UDP is unreliable, the applications may support the reliability themselves through retransmission. This reliability guarantee would lead to the interruption of network communication if UDP packets are lost upon restoration. In conclusion, on constructing the RDG, we employ TCP as well as UDP packets whose source and destination are both the VMs within the VMC.

### 5.2 Optimizations on Restore

HotRestore adopts several optimizations to reduce the restore latency as well as performance overhead.

**Sequential loading of working set pages.** In HotRestore, the working set pages scatter here and there in the snapshot file and are mixed with the pages saved in background. Upon restoration, HotRestore dequeues the guest frame number (gfn) from the FAFL queue to index the working set page. However, the gfn order in the FAFL queue is not consistent with that in the snapshot file. For example, page *A* is firstly saved in background, and then page *B* is saved and recorded due to memory write operation, finally *A* is recorded due to memory read. In this case, *A* is stored in front of *B* in the snapshot file, but is behind *B* in the FAFL queue. Fetching the working set pages in FAFL order would seek the pages back and forth in the disk file, thereby incurring longer latency. To solve this problem, we rearrange the working set pages by their file offset once the revised size $S^*$ is known, so that the pages can be loaded sequentially from the snapshot file. Besides, the pages that are neighboring in the snapshot file can be loaded together to further reduce the amount of file read operations.

**DMA cluster.** DMA pages are touched frequently for IO intensive workloads, e.g., we observe about 280,000 DMA access in 30 seconds execution under Gzip for a VM configured with 2GB RAM. DMA page access exhibits two characteristics: First, the page will be accessed repeatedly (may be hundreds or even thousands times), therefore the amount of touched pages is actually only a few hundreds or thousands. Second, the accessed pages are always neighboring. These observations inspire us to load multiple neighboring pages (HotResotre loads 4 pages) for each on-demand DMA paging to reduce the occurrence of demand-loading, thereby avoiding significant performance degradation incurred by page faults.

## 5.3 Real World I/O Bandwidth

In real world scenarios, the restore latency is related to not only the working set size, but also the available I/O bandwidth. The snapshot procedures may contend for I/O bandwidth, making the latencies various even if the file sizes are identical. As a result, the practical start times may be not as indicated in the restore line. In our test environments, the working set size is small while the bandwidth is sufficient, so that the restore latency is less affected. However, we believe that this problem will get worse for memory intensive workloads, especially in I/O intensive data centers. VM placement [30] and I/O schedule layer [32] are alternative approaches to mitigate this problem, and we leave this as our future work.

## 5.4 Non-deterministic Events

There exist many events that lead to non-deterministic system execution, they fall into two categories: external input and time [13]. The external input involves the data sent from another entity, such as the network packets from the web server, or user operations (e.g., booting a new application). Time refers to the point in the execution stream where the internal or external event takes place, for example, the receiving time of network packets. The combination of the two influences the results of several strategies resides in CPU scheduler, IO scheduler, and TCP/IP stack, so that the system execution is diverged even the system is restored from the same snapshot point. Fortunately, compared to the individual desktop which involves multiple tasks and continual interactive operations, the distributed applications running in the virtual machine cluster are always monotonous and involves less user interaction, making the execution of the virtual machine be always deterministic.

## 6 Evaluation

We conduct the experiments on eight physical servers, each configured with 8-way quad-core Intel Xeon 2.4GHz processors, 48GB DDR memory and Intel 82576 Gbps Ethernet card. The servers are connected via switched Gbps Ethernet. We configure 2GB memory for the VMs. The operating system on physical server and virtual machine is debian6.0 with 2.6.32-5-amd64 kernel. We save the snapshot files in local disk, and then restore the virtual machines from snapshot files to evaluate HotRestore.

## 6.1 Elastic Working Set

We evaluate the elastic working set in terms of hit rate, accuracy, working set size and scalability, under several applications. The applications include: 1) Compilation, a development workload which involves memory and disk I/O operations. We compile the Linux 2.6.32-5 kernel. 2) Gzip is a compression utility, we compress the */home* directory whose size is 1.4GB. 3) Mummer is a bioinformatics program for sequence alignment, it is CPU and memory intensive [5]. We align two genome fragments obtained from NCBI [2]. 4) Pi calculation, a CPU intensive scientific program. 5) MPlayer is a movie player. It prefetches a large fraction of movie file into buffer for performance requirements. 6) MySQL is a database management system [6], we employ SysBench tool [9] to read (write) data from (to) the database. We conduct the experiments ten times and report the average as well as the standard deviation.

### 6.1.1 Hit Rate and Accuracy

We first measure the hit rate and accuracy under different working set sizes. We start the VM after the working set in the specified size is loaded, and disable the background load thread to trace all memory accesses. If the traced page is in the working set, then a hit occurs, and the hit count increases by 1. The traced page will not be traced again, since multiple hits on the same page would increase the hit rate and accuracy. Once the count of the traced pages reaches the given size, we calculate the hit rate by the ratio of hit count to the given size. The accuracy calculation is a little different. We observe that most of the untouched working set pages will be touched within a short period. Therefore, we trace an extra 1/5 size, and calculate the accuracy by the ratio of hit count to 1.2 times given size.

Table 1 demonstrates the hit rate and accuracy of FAFL compared to LRU and CLOCK under Linux kernel compilation. It can be seen that the hit rate with FAFL is higher, e.g., for a 15K size, the hit rate with FAFL

| | Hit Rate | | | Accuracy | | |
|---|---|---|---|---|---|---|
| Size | **FAFL** | **LRU** | **CLOCK** | **FAFL** | **LRU** | **CLOCK** |
| 5K | 0.816 | 0.778 | 0.814 | 0.859 | 0.817 | 0.838 |
| 10K | 0.845 | 0.875 | 0.749 | 0.945 | 0.918 | 0.926 |
| 15K | 0.944 | 0.857 | 0.868 | 0.952 | 0.954 | 0.952 |
| 17K | 0.912 | 0.918 | 0.822 | 0.958 | 0.955 | 0.955 |
| 20K | 0.889 | 0.888 | 0.828 | 0.963 | 0.962 | 0.923 |
| 25K | 0.870 | 0.861 | 0.869 | 0.962 | 0.963 | 0.970 |

Table 1: Hit rate and accuracy for working set with various sizes. Here, the size refers to page count, the calculated $WSS$ is 17K. STDEV is minor and thus is removed.

| | Hit Rate | | | Accuracy | | |
|---|---|---|---|---|---|---|
| Workloads | **FAFL** | **LRU** | **CLOCK** | **FAFL** | **LRU** | **CLOCK** |
| Gzip | 0.806 | 0.768 | 0.883 | 0.974 | 0.979 | 0.966 |
| MySQL | 0.947 | 0.655 | 0.912 | 1 | 0.998 | 1 |
| Mummer | 0.931 | 0.835 | 0.812 | 1 | 0.971 | 0.909 |
| Pi | 0.628 | 0.562 | 0.589 | 0.702 | 0.682 | 0.793 |
| MPlayer | 0.890 | 0.825 | 0.862 | 0.926 | 0.923 | 0.892 |

Table 2: Hit rate and accuracy under various workloads. STDEV is minor and thus is removed.

is 94.4% while it is 85.7% with LRU and 86.8% with CLOCK. The improvement is mainly contributed to the FAFL queue which captures the access order of VM execution more accurately. An interesting result is that the hit rate decreases as the size grows, e.g., the hit rate with FAFL decreases from 94.4% to 88.9% while the size increases from 15K to 20K. We suspect that this is because the execution suffers from larger deviation after longer execution time. Despite this, the hit rate is still high. Besides, the accuracy of the three manners exceeds 95% in most cases. This fact proves that the memory tracing method during *post-snapshot* could capture the memory accesses accurately.

We also measure the hit rate and accuracy under other workloads. Table 2 illustrates the results when the calculated working set size is applied upon restoration. We can see that the hit rate under Gzip workload is low, this is because Gzip is I/O intensive and involves large amounts of DMA operations which always diverge after the VM is restored. For MySQL, Mummer and MPlayer workloads, the hit rate and accuracy is high due to two reasons: i) these workloads consist of a large amount of pages that hit the buffer cache which facilitate working set estimation and ii) their executions are almost deterministic. The high rate under Pi workload is poor, e.g., it is 62.8% with FAFL, this is due to the dynamic memory allocation during execution. Fortunately, the associated working set size is small, it is less than 1K pages, so that the performance loss is insignificant after the VM is restored. These three methods all achieve high accuracy, this means that most of the loaded working set pages will be touched by the applications after the VM is restored. The accuracy under Pi workload is low, we guess that this is due to the non-deterministic execution of Pi. On average, FAFL increases the hit rate by 15.3% and 4.92% respectively compared to LRU and CLOCK.

### 6.1.2 Working Set Size

Here, the working set size is referred to as the total amount of loaded state including the CPU state, devices' state, page content and extra information such as page address. Zero page compression used in most snapshot technologies may achieve 50% reduction of snapshot size [21]; however, the reduction is specific to workload and relates to application execution time. As a result, this optimization is disabled in this experiment. Table 3 compares HotRestore with *working set restore* [39] which calculates the working set size through the statistical sampling approach.

As expected, HotRestore loads less pages upon restoration. Compared to *working set restore*, HotRestore reduces the working set size by 50.95% on average, therefore the restore latency is supposed to be halved accordingly. Although restoring a VM requires extra time to sort the working set pages and set protection flags, the extra time is minor. Compared to the default restore method in QEMU/KVM which takes about 60 seconds to load a 2G snapshot file, HotRestore can restore the VM within 3 seconds for most workloads.

| Modes | **Compilation** | **Gzip** | **Mummer** | **Pi** | **MPlayer** | **MySQL** |
|---|---|---|---|---|---|---|
| HotRestore | 72.0(2.88) | 60.9(15.3) | 347.5(30.7) | 1.5(0.09) | 37.2(6.4) | 42.4(8.9) |
| Working Set Restore | 153.0(7.92) | 113.5(21.2) | 836.3(117.9) | 2.76(0.17) | 75.3(9.4) | 87.8(11.4) |
| **Reduction** | 52.94% | 46.34% | 58.45% | 45.65% | 50.6% | 51.71% |

Table 3: Comparison of working set sizes (MB). STDEV is also reported

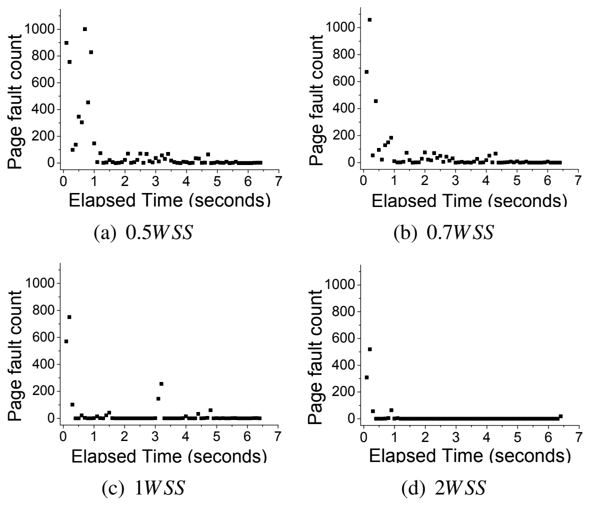
### 6.1.3 Scalability

The results in Table 1 have shown that the hit rate and accuracy remain high regardless of the working set sizes, they therefore prove that the elastic working set can scale up or scale down without compromising the hit rate or accuracy.

On the other hand, however, scaling down the size would bring more page faults due to demand-paging and thus imposes performance overhead. Therefore, the performance loss or the count of page faults should be measured and reported. We trace the page faults after restoration and record the count in 100ms interval. Figure 6 shows the count on different working set sizes: the calculated $WSS$ and its two variants, $0.5WSS$ and $0.7WSS$. $WSS$ is 18327 pages in this experiment. As we can see, the decrease of the working set size indeed incurs more page faults. Specifically, the numbers are 2046, 3539 and 5690 for $WSS$, $0.7WSS$ and $0.5WSS$ respectively during the 7 seconds tracing period. Intuitively, the increased count of page faults should be equivalent to or approximate the saved page count upon restoration. However, our results show that it is not. For example, $0.7WSS$ loads 5498 less pages upon restoration but incurs only 1493 more page faults compared to $WSS$. This is mainly because the pages stored in the FAFL queue depict the page access order. If the $WSS$ applied upon restoration is less than the calculated $WSS$, the remaining working set pages tend to be dequeued and loaded by the background load thread. The FAFL method effectively reduces the count of page faults compared to the approach which loads pages blindly.

The result for $2WSS$ is also given, as shown in Figure 6(d). It can be seen that the count of page faults can be further reduced due to the increase of $WSS$, specifically, it decreases to 958. However, we believe that the reduction is insignificant. This is because that one page fault incurs about 200us interruption which involves the time to exit to VMM, load the page from local disk, remove protection flag and resume the VM. Therefore, the overall overhead for handling thousands of page faults is negligible compared to the gain of 50% reduction on the working set size or namely the restore latency.

These results show that the working set can scale up/down without significant performance loss. This gives us a hint that slight working set size revision is allowable when restoring the VMC. The results under other workloads are similar, so that they are ignored due to space constraints.



(a) $0.5WSS$ (b) $0.7WSS$ (c) $1WSS$ (d) $2WSS$

Figure 6: Comparison of page fault count after restore.

## 6.2 Restoration of VMC

In this section, we evaluate HotRestore in the VMC and discuss the restore latency as well as the TCP backoff duration. The native restore method of QEMU/KVM incurs dozens of seconds disruption, so the result is neglected here. We conduct the experiments with three restore mechanisms:

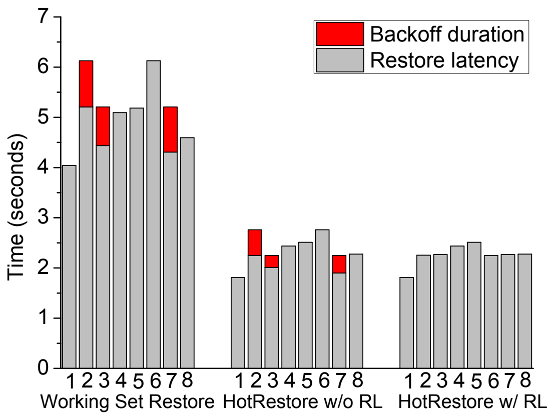
**Working Set Restore.** It reduces the restore latency by prefetching the working set for a single VM.

**HotRestore without Restore Line (HotRestore w/o RL).** It reduces the restore latency through elastic working set, but does no further work on VMC restore.

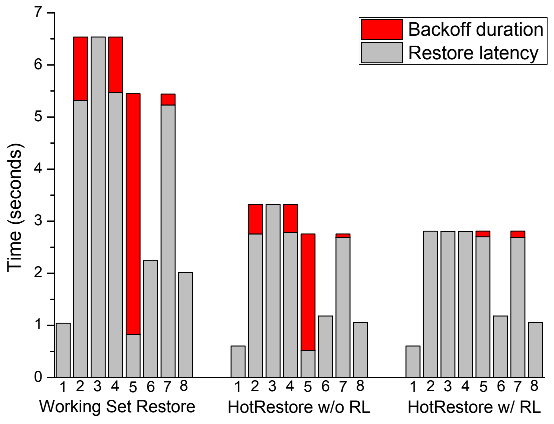
**HotRestore with Restore Line (HotRestore w/ RL).** It exploits the elastic working set and utilizes restore line to reduce the TCP backoff duration.

### 6.2.1 Detailed View of Disruption

We first illustrate the disruption duration consisting of restore latency and TCP backoff duration in detail. We setup the VMC with 8 VMs, and apply two representative network applications: i) Distcc [3] which is a com-



(a) Disruption under Distcc



(b) Disruption under Elasticsearch

Figure 7: Comparison of disruption time in the VMC.

pilation tool that adopts client/server architecture to distribute the tasks across the nodes, and ii) Elasticsearch [4] which is a distributed, de-centralized search server used to search documents. The nodes play equivalent role in Elasticsearch. The practical TCP backoff duration is hard to depict unless modifying the guest OS, therefore, we approximate the value by the difference between start times of communicating VMs. If the VM is communicating with multiple VMs, the maximum time difference will be adopted as the duration. We use NTP to synchronize the times of physical server, so that the difference among physical times of servers is within only a few milliseconds.

Figure 7 compares the detailed disruption duration of VMs of HotRestore to that of Working Set Restore. The gray bar illustrates the restore latency, while the red bar depicts the TCP backoff duration. It can be seen that the restore latencies of VMs are various and thus cause the backoff problem. Take Distcc as an example, the latencies of VM2 and VM6 are 5.21 seconds and 6.13 seconds respectively. Since VM2 depends on VM6, hence the backoff duration of VM2 is 0.92 seconds. It can be seen that the restore latencies of VMs with Working Set Restore are much longer than that with HotRestore methods. The HotRestore w/o RL method reduces the restore latency by employing a smaller working set size, so that the backoff duration decreases accordingly. The results in Figure 7(a) show that the HotRestore w/o RL reduces the disruption by 54.1% on average compared to Working Set Restore. The HotRestore w/ RL method reduces the disruption further. Although there is no significant decrease on restore latency compared to HotRestore w/o RL, the backoff is eliminated as a result of the restore line. Generally, it achieves a 56.7% reduction on disruption duration compared to Working Set Restore under Distcc workload.

The disruption under Elasticsearch workload shows similar results. Compared to Working Set Restore, HotRestore w/o RL reduces the average disruption duration by 48.6%. It is worth noting that the TCP backoff appears in HotRestore w/ RL, e.g., the backoff duration of VM5 and VM7 are 0.1 and 0.12 seconds respectively. This is because VM4, VM5 and VM7 form a dependency ring and are given identical working set size to be started simultaneously. However, due to the fluctuation of disk IO speed, the practical start times of VMs are different. In this experiment, VM5 and VM7 start earlier than VM4, as a result, they suffer from TCP backoff problem. On average, HotRestore w/ RL reduces the disruption by 53.6% compared to Working Set Restore.

### 6.2.2 Details on TCP Backoff Duration

The above experiments present an overview of backoff duration. We can see that some VMs do not experience TCP backoff, while others suffer from long backoff duration, e.g., the duration of VM5 is 4.49 seconds under Elasticsearch. In this section, we will exhibit the details on backoff duration, especially for the complicated network topology as the VMC scales out. We evaluate HotRestore for a VMC configured with 8, 12, 16 VMs under Elasticsearch workload, and reports the details on VMC backoff duration. There is no direct metric to measure the backoff duration for the whole VMC, therefore, we calculate all the backoff duration between each two communicating VMs, and report the maximum, minimum, average and median value.

Figure 8 demonstrates the results on backoff duration. As expected, HotRestore w/ RL achieves the least duration. Compared to Working Set Restore which incurs 2.66 seconds backoff duration on average, HotRestore w/ RL reduces the average duration to less than 0.07 seconds. As explained earlier, the duration with HotRestore is mainly due to the existence of dependency ring as well as the difference of VMs' start times. Besides, we can

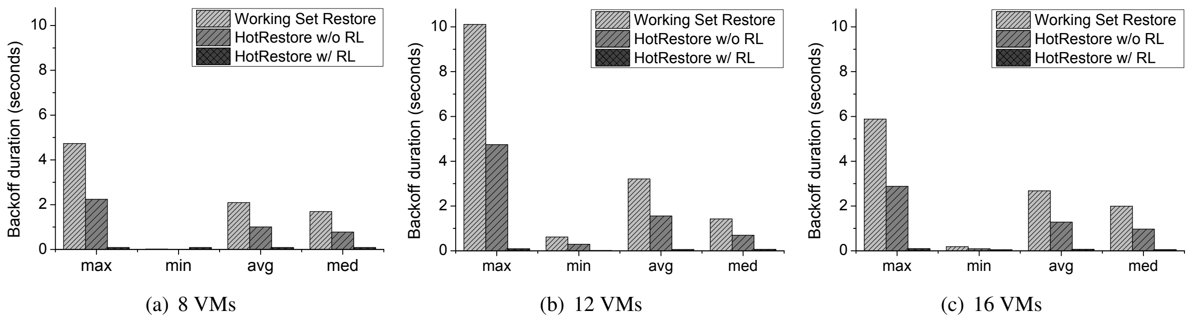


Figure 8: Comparison of TCP backoff duration.

see that the maximum backoff duration with Working Set Restore exceeds 10.1 seconds in Figure 8(b). The long duration may make the application inside the VM experience a long freeze stage even if the VM actually has already started. HotRestore solves the problem through the restore line. As we can see, even the maximum duration is less than 0.14 seconds in HotRestore. As a result, the VM after restoration is able to execute without perceivable interruption.

These results demonstrate that HotRestore w/ RL reduces the TCP backoff duration to milliseconds and scales well for larger scale VMC. Besides, for low latency network where the proposed restore protocol is not suitable, the HotRestore w/o RL approach can still work and bound the duration in a few seconds.

It is worth noting that the TCP backoff duration here is simply calculated by the difference between the start times of communicating VMs. In practice, however, the backoff duration increases twofold for each timeout, making the practical backoff duration of the Working Set Restore be much larger than the value shown in Figure 8. In other words, HotResotre would performs much better than Working Set Restore in practical scenarios.

## 6.3 Performance Overhead

This part will measure the incurred performance overhead of HotRestore. The overhead mainly comes from two aspects. One is the overhead on snapshot. Compared to traditional snapshot that only traces write operations, HotRestore traces both read and write operations, so that it incurs extra overhead. Another one is the overhead after restoration. The demand-paging will trigger page faults, making the VM exit to VMM for loading the desired page from disk file.

### 6.3.1 Overhead during Snapshot

We measure the overhead in terms of the count of traced pages during snapshot. Table 4 compares the results of HotSnap (as the baseline shows) and HotRestore. It can be seen that the increase incurred by HotRestore ranges from 11.3% to 124%.

Upon a page fault triggered by access tracing, the VMM suspends the VM and handles the page fault. The overhead to handle the fault mainly consists of two parts. First, the VM exits to VMM which removes the read/write protect flag and then resumes the VM. The exit and entry of VM to VMM takes 38us in our platform. Second, the VMM saves the traced page into storage for memory write operation. Saving one page (4K) takes about 150us on average. Fortunately, tracing the memory read operations in HotRestore does not require saving the page. As a result, the extra time during snapshot creation incurred by tracing read operations is minor. Table 5 compares the snapshot creation time under various workloads. As an example, the total time to save the VM state in HotRestore increases by 1.3 seconds compared to that of Baseline under Compilation workload.

| Modes | **Compile** | **Gzip** | **Pi** | **MPlayer** | **MySQL** |
|---|---|---|---|---|---|
| Baseline | 25933 | 53447 | 1523 | 21510 | 12825 |
| HotRestore | 34348 | 59466 | 3413 | 30217 | 17598 |
| **Increase** | 32.4% | 11.3% | 124% | 40.4% | 37.2% |

Table 4: Comparison of traced page count.

| Modes | **Compile** | **Gzip** | **Pi** | **MPlayer** | **MySQL** |
|---|---|---|---|---|---|
| Baseline | 85.3 | 79.5 | 54.2 | 72.5 | 77.3 |
| HotRestore | 86.6 | 81.1 | 54.4 | 74.2 | 78.2 |
| **Increase** | 1.3 | 1.6 | 0.2 | 1.7 | 0.9 |

Table 5: Comparison of snapshot duration (seconds).

### 6.3.2 Overhead after Restoration

We setup Elasticsearch in the VMC configured with 8 VMs. We measure the performance loss in terms of the response time. Specifically, we fill Elasticsearch with 1

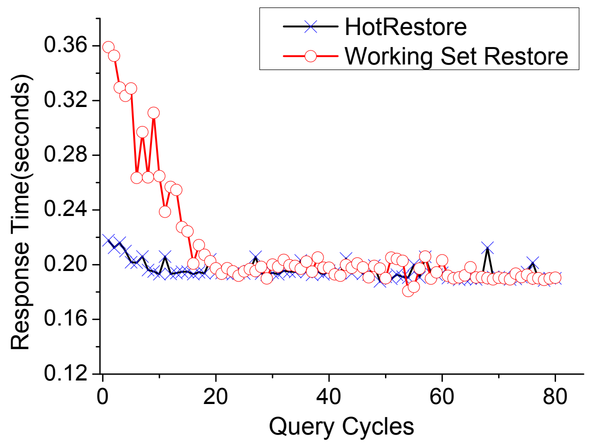


Figure 9: Comparison of response time.

million micro-blogs and launch ten threads to query concurrently. Each query requests different keywords and acquires 40 micro-blogs. The average response time for each query is about 0.192s during normal execution.

Figure 9 demonstrates the average response time of ten threads for continuous queries. As we can see, the response time with Working Set Restore is long after restoration. Specifically, the latency of the first query is 0.36 seconds, which is about twice the latency of normal execution. The reason is as follows: The Elasticsearch node will handle the query after the associated VM starts. However, some other nodes are still suspended to load the working set, thereby causing the backoff among these nodes. As a result, the requested node cannot reply as fast as that in normal execution with fewer peers, especially for concurrent queries from ten threads. The response time decreases for the subsequent queries, due to the coordination with peers that are resumed recently. HotRestore shows no significant performance loss. The requested node can coordinate with peers within a short period after restoration due to the short backoff duration. In practice, the average response time with HotRestore is 0.215 seconds for the first six queries, only a little higher than that of normal execution.

The response time with HotRestore returns to the normal value from the 7th cycle, while it keeps high until the 16th cycle with the Working Set Restore approach. This implies that HotRestore halves the time for Elasticsearch to regain the full capacity. Although the working set sizes of some certain VMs are revised to be smaller and thus degrade the performance for the single VM, the overall performance of entire VMC is improved due to the elimination of the network interruption.

## 7 Related Work

### 7.1 VM Restoration

The idea of fast restore is not new, several approaches have been proposed to fast restore (or start) the processes and the operating systems. Recovery Oriented Computing (ROC) [33] achieves fast recovery of process upon failures by fine grained partitioning and recursive restart. Li et al. [29] track the pages touched by applications during post-checkpoint, and use these touched pages to restart the processes fast. Windows adopts SuperFetch [8] and ReadyBoost [7] to accelerate the application and OS launch times respectively by monitoring and adapting the usage patterns and prefetching the frequently used files and data into memory so that they can be accessed quickly when needed. HotRestore is fundamentally different from these works in that it focuses on the restore latency of virtual machines, rather than processes or operating systems.

There is not much work on improving VM restore. The most simple approach is *eager restore*, which starts the virtual machine after all the state including device state and memory state are loaded [37]. This approach, obviously, incurs long latency for VMs equipped with large size memory. *Lazy restore* [18] reduces the latency to milliseconds through starting the VM after CPU state and device state are loaded and loading the memory state in an on-demand way. It however incurs serious performance loss due to large amounts of demand-paging after restore, especially in the beginning execution after the VM is rollbacked. *Working set restore* [39] addresses the performance issue by prefetching the working set upon restore, at the cost of only a few seconds downtime. Our work on a single VM restore shares a similar philology to *working set restore*. The difference is, their method employs working set to reduce the time-to-responsiveness metric, yet we propose an elastic working set for restoring the virtual machine cluster.

### 7.2 VMC Restoration

There have been amounts of work about restoring a distributed system, and the key of them is to guarantee the consistency of the saved state. Several work create the global consistent state through coordinated snapshotting approach, so that the system can be rolled back directly from the saved state [27, 20, 12]. Another field assumes that the nodes create snapshots independently, therefore they focus on guaranteeing the global consistency among snapshots upon restore and avoiding domino effect [16, 36, 34]. Several recently proposed snapshot systems for virtual machine cluster create the consistent global state [25, 11, 19] when snapshotting, but make no improvement on restoration. We guarantee the consistency while the snapshots are being created by our prior work called HotSnap [14], and our concern within the paper is to reduce the restore latency as well as the TCP backoff duration.

Besides, the TCP backoff problem incurred during snapshot has received attention recently. VNSnap [25] saves the memory state in a temporary memory region and then flushes into disk asynchronously, thereby reducing the discrepancy of snapshot completion times to reduce the backoff duration. Emulab [11] addresses this issue by synchronizing clocks across the nodes to suspend all nodes for snapshot simultaneously. Our work emphasizes on backoff duration upon restoration. By exploiting the network causal order, we propose a restore line method to minimize the backoff duration.

### 7.3 Working Set

Working set captures the notion about memory access locality. Denning describes which pages should be swapped in and out of the working set [15]. LRU [31], CLOCK [23] and CAR [10] are improved methods to identify which pages should be replaced into the working set. HotRestore adopts the FAFL queue to record the traced pages as candidate working set pages, and then produces the working set according to the desired size. Besides, several systems adopt the working set to achieve fast restoration of process [29] or VM [39]. They employ memory tracing to capture the working set pages and leverage the sampling approach [38] to estimate the working set size. HotRestore is different in that it figures out a hybrid size based on sampling during normal execution as well as statistic during snapshot, and then halves the size as the expected working set size. The size shrink does not impose significant performance loss due to the well scalability of the elastic working set.

## 8 Limitations and Future Work

There still exist several limitations in HotRestore. First, the proposed elastic working set would perform poor for non-deterministic applications, especially for applications in SMP virtual machines due to the variable vCPU scheduling. Other non-deterministic events, as described in §5.4, will also lead to the divergence of VM execution after restoration. As a result, the on-demand paging would occur frequently, and further imposes significant performance loss. Second, for some applications, e.g., long running scientific programs, where fast restore is inessential, HotRestore may incur unnecessary overhead due to extra read traces during snapshot, given that the VMC snapshot is frequently created.

Therefore, our ongoing work contain two directions. The first is to analyze the memory traces for non-deterministic applications in SMP (Symmetric Multi-Processing) VM for seeking a suitable page replacement algorithm to build an accurate working set, with the aim to reduce the amount of page faults as well as eliminating performance degradation. What's more, given that snapshot is required more frequently than restore, we plan to make a holistic study on performance overhead with multiple snapshots along with one restore operations in real world scenarios, with the aim to find an adaptive snapshot/restore policy to minimize the overall overhead for long running applications.

## 9 Conclusions

In this paper, we present HotRestore, a restore system which enables fast restore of the virtual machine cluster without perceivable disruption. HotRestore employs an elastic working set to reduce the restore latency without compromising the application performance, and proposes restore line to reduce the TCP backoff duration. The key insight in restore line is that the start times of VMs can be revised to match the network causal order of VMs. We have implemented HotRestore on QEMU/KVM platform. Our evaluation results show that the whole VMC can be restored within a few seconds, what's more, the applications can regain the full capacity rapidly after restoration benefiting from the elimination of TCP backoff. We believe that HotRestore will help improve system reliability and performance after failure recovery, especially in the scenarios where failures and restoration are requied frequently.

## Acknowledgements

We would like to thank our shepherd Theophilus Benson and the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by China 973 Program under Grant No. 2011CB302602, China 863 Program under Grant No. 2011AA01A202 and 2013AA01A213, HGJ Program under Grant No. 2010ZX01045-001-002-4 and 2013ZX01039-002-001-001, NSFC Program under Grant No. 61202424, 91118008, and 61170294, and Fundamental Research Funds for the Central Universities under Grant No.YWF-14-JSJXY-017. Jianxin Li and Tianyu Wo are the corresponding authors of this paper.

## References

[1] Amazon ec2. `Http:// aws.amazon.com/ec2/`.

[2] National center for biotechnology information. ftp://ftp.ncbi.nih.gov/.

[3] Distcc. http://code.google.com/p/distcc/.

[4] Elasticsearch. http://www.elasticsearch.org/.

[5] Mummer. http://mummer.sourceforge.net/.

[6] Mysql. http://www.mysql.com/.

[7] Readyboost. http://en.wikipedia.org/wiki/ReadyBoost.

[8] Superfetch. http://en.wikipedia.org/wiki/Windows_Vista_I/O_technologies.

[9] Sysbench. http://sysbench.sourceforge.net/.

[10] S. Bansal and D. S. Modha. Car: Clock with adaptive replacement. In *Proceedings of USENIX FAST*, pages 187–200, 2004.

[11] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of EuroSys*, pages 173–186, 2009.

[12] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3 (1):63–75, 1985.

[13] P. M. Chen and B. D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Proceedings of the HotOS*, pages 133–138, 2001.

[14] L. Cui, B. Li, Y. Zhang, and J. Li. Hotsnap: A hot distributed snapshot system for virtual machine cluster. In *Proceedings of USENIX LISA*, pages 59–73, 2013.

[15] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[16] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[17] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of OSDI*, pages 1–14, 2010.

[18] R. Garg, K. Sodha, and G. Cooperman. A generic checkpoint-restart mechanism for virtual machines. In *CoRR abs/1212.1787*, 2012.

[19] R. Garg, K. Sodha, Z. Jin, and G. Cooperman. Checkpoint-restart for a network of virtual machines. In *IEEE International Conference on Cluster Computing*, pages 1–8, 2013.

[20] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, pages 144–154, 1991.

[21] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Communications of ACM*, 53 (10):85–93, 2010.

[22] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference*, pages 113–128, 2008.

[23] S. Jiang, F. Chen, and X. Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of ATC*, pages 323–336, 2005.

[24] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. In *Parallel and Distributed Processing and Applications*, pages 937–946, 2005.

[25] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimmal downtime. In *Proceedings of DSN*, pages 87–98, 2011.

[26] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of STOC*, pages 302–311, 1984.

[27] J. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955–960, 1993.

[28] A. Kivity, Y. Kamay, D. Laor, and U. Lublin. Kvm: the linux virtual machine monitor. *Computer and Information Science*, 1:225–230, 2007.

[29] Y. Li and Z. Lan. A fast restart mechanism for checkpoint/recovery protocols in networked environments. In *Proceedings of DSN*, pages 217–226, 2008.

[30] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of INFOCOM*, pages 1–9, 2010.

[31] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD*, pages 297–306, 1993.

[32] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling io in virtual machine monitors. In *Proceedings of VEE*, pages 1–10, 2008.

[33] D. Patterson, A. Brown, P. Broadwell, and et al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. In *Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science*, 2002.

[34] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, 1980.

[35] B. Schroeder and G. A. Gibson. Understanding fail-

ures in petascale computers. *Journal of Physics*, 78: 1–11, 2007.

[36] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transations on Computer Systems*, 3(3):204–226, 1985.

[37] G. Vallee, T. Naughton, H. Ong, and S. L. Scott. Checkpoint/restart of virtual machines based on xen. In *Proceedings of HAPCW*, 2006.

[38] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of USENIX OSDI*, pages 181–194, 2002.

[39] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of VEE*, pages 534–533, 2009.

[40] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing vm checkpointing for restore performance in vmware esxi. In *Proceedings of USENIX ATC*, pages 1–12, 2013.

[41] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *Proceedings of USENIX ATC*, pages 17–22, 2011.

# Compiling Abstract Specifications into Concrete Systems – Bringing Order to the Cloud

Ian Unruh
*Kansas State University*
*iunruh@ksu.edu*

Alexandru G. Bardas
*Kansas State University*
*bardasag@ksu.edu*

Rui Zhuang
*Kansas State University*
*zrui@ksu.edu*

Xinming Ou
*Kansas State University*
*xou@ksu.edu*

Scott A. DeLoach
*Kansas State University*
*sdeloach@ksu.edu*

## Abstract

Currently, there are important limitations in the abstractions that support creating and managing services in a cloud-based IT system. As a result, cloud users must choose between managing the low-level details of their cloud services directly (as in IaaS), which is time-consuming and error-prone, and turning over significant parts of this management to their cloud provider (in SaaS or PaaS), which is less flexible and more difficult to tailor to user needs. To alleviate this situation we propose a high-level abstraction called the *requirement model* for defining cloud-based IT systems. It captures important aspects of a system's structure, such as service dependencies, without introducing low-level details such as operating systems or application configurations. The requirement model separates the cloud customer's concern of *what* the system does, from the system engineer's concern of *how* to implement it. In addition, we present a "compilation" process that automatically translates a requirement model into a concrete system based on pre-defined and reusable knowledge units. When combined, the requirement model and the compilation process enable repeatable deployment of cloud-based systems, more reliable system management, and the ability to implement the same requirement in different ways and on multiple cloud platforms. We demonstrate the practicality of this approach in the *ANCOR* (Automated eNterprise network COmpileR) framework, which generates concrete, cloud-based systems based on a specific requirement model. Our current implementation[1] targets OpenStack and uses Puppet to configure the cloud instances, although the framework will also support other cloud platforms and configuration management solutions.

[1]Current ANCOR implementation is available and is distributed under the GNU (version 3) General Public License terms on: https://arguslab.github.io/ancor/

**Tags**: cloud, modeling networking configuration, configuration management, deployment automation

## 1 Introduction

Cloud computing is revolutionizing industry and reshaping the way IT systems are designed, deployed and utilized [3]. However, every revolution has its own challenges. Already, companies that have moved resources into the cloud are using terms like "virtual sprawl" to describe the mess they have created [38]. Cloud services are currently offered in several models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). While these options allow customers to decide *how much* management they want to perform for their cloud-based systems, they do not provide good *abstractions* for effectively managing those systems or addressing diverse user needs.

IaaS solutions such as Amazon Web Services (AWS) and OpenStack allow cloud users to access the raw resources (compute, storage, bandwidth, *etc.*); however, it forces users to manage the software stack in their cloud instances at a low level. While this approach gives users tremendous flexibility, it also allows the users to create badly configured or misconfigured systems, raising significant concerns (especially related to security) [5, 7]. Moreover, offering automatic scalability and failover is challenging for cloud providers because replication and state management procedures are application-dependent [3]. On the other hand, SaaS (also known as "on-demand software") provides pre-configured applications to cloud users (*e.g.*, SalesForce and Google Apps). Users typically choose from a set of predefined templates, which makes it difficult to adequately address the range of user needs. PaaS (*e.g.*, Google App Engine, Heroku, and Windows Azure) is somewhere in the middle, offering computing platforms with various pre-installed operating systems as well as services and allowing users to deploy their own applications as well.

As PaaS is a compromise between IaaS and SaaS, it also inherits the limitations of both to various degrees. For example, users can be easily "locked in" to a PaaS vendor, like in SaaS, and the configuration of applications is still on the users' shoulders, like in IaaS.

We observe that existing cloud service models suffer from the lack of an appropriate *higher-level abstraction* capable of capturing objectives and functionality of *the complete IT system*. Such an abstraction, if designed well, can help both the creation and the long-term maintenance of the system. While there have been attempts at providing abstractions at various levels of cloud-based services, none have provided an abstraction that both separates user requirements from low-level platform/system details and provides a global view of the system. This has limited the usefulness of those solutions when it comes to long-term maintenance, multi-platform support, and migration from one cloud provider to another. We believe to be effective, the *abstraction* should exhibit the following properties.

1. It must be capable of representing *what* a user needs instead of low-level details on *how* to implement those needs. A major motivation for using cloud infrastructures is to outsource IT management to a more specialized workforce (called *system engineers* hereafter). Communicating *needs* from users to engineers is better served using higher-level abstractions as opposed to low-level system details.

2. It must support automatic compilation into valid concrete systems on different cloud infrastructures. Such compilation should use well-defined knowledge units built by the system engineers and be capable of translating a specification based on the abstraction (*i.e.*, an *abstract specification*) into different concrete systems based on low-level implementation/platform choices.

3. It should facilitate the long-term maintenance of the system, including scaling the system up/down, automatic fail over, application update, and other general changes to the system. It should also support orchestrating those changes in a secure and reliable manner and aid in fault analysis and diagnosis.

We believe such an abstraction will benefit all three existing cloud service models. For IaaS, an abstract specification will act as a common language for cloud users and system engineers to define the system, while the compilation/maintenance process becomes a tool that enables system engineers to be more efficient in their jobs. Re-using the compilation knowledge units will also spread the labor costs of creating those units across a large number of customers. In the SaaS model the system engineers will belong to the cloud provider so the abstract specification and the compilation/maintenance process will help them provide better service at a lower cost. In the PaaS model we foresee using the abstraction and compilation process to stand up a PaaS more quickly than can be done today. This could even foster the convergence to a common set of PaaS APIs across PaaS vendors to support easier maintenance and migration between PaaS clouds.

There are multiple challenges in achieving this vision. The most critical is whether it is feasible to design the abstraction so that it can capture appropriate system attributes in a way that is meaningful to users and system engineers while being amenable to an automated compilation process that generates valid concrete systems.

The ***contributions*** of our work are:

- We design an abstract specification for cloud-based IT systems that separates user requirements from system implementation details, is platform-independent, and can capture the important aspects of a system's structure at a high level.

- We design a compilation process that (1) translates the high-level specification into the low-level details required for a particular choice of cloud platform and set of applications, and (2) leverages a mature configuration management solution to deploy the resulting system to a cloud.

- We show that maintaining an abstract specification at an appropriate level enables users to address automatic scaling and failover even though these processes are highly application-dependent, and supports a more reliable and error-free orchestration of changes in the system's long-term maintenance.

To demonstrate the efficacy of our approach, we implemented and evaluated a fully-functional prototype of our system, called *ANCOR* (Automated eNterprise network COmpileR). The current implementation of ANCOR targets OpenStack [45] and uses Puppet [20] as the configuration management tool (CMT); however, the framework can also be targeted at other cloud platforms such as AWS, and use other CMT solutions such as Chef [34].

The rest of the paper is organized as follows. Section 2 explains the limitations of current solutions as well as the enabling technologies used in this work. Section 3 presents an overview of the framework, including the proposed abstraction and the compilation workflow. Section 4 describes the implementation of the ANCOR framework and its evaluation. We discuss some other relevant features of the approach and future work in Section 5, followed by related work and conclusion.

## 2 Background

### 2.1 Limitations of Available Automation and Abstraction Technologies

Recent years have seen a proliferation of cloud management automation technologies. Some of these solutions (*e.g.,* AWS OpsWorks) tend to focus on automation as opposed to abstraction. They include scripts that automatically create virtual machines, install software applications, and manage the machine/software lifecycle. Some are even able to dynamically scale the computing capacity [36, 38, 39]. Unfortunately, none of these solutions provide a way to explicitly document the dependencies between the deployed applications. Instead, dependencies are *inferred* using solution-specific methods for provider-specific platforms. Not only is this unreliable (*e.g.*, applications may have non-standard dependencies in some deployments), but it lacks the capability to *maintain* the dependency after the system is generated. Ubuntu Juju [41] is a special case that is described and discussed in Section 6 (Related Work).

Recent years have also seen a general movement towards more abstractions at various levels of cloud services, especially in PaaS. Examples include Windows Azure Service Definition Schema (.csdef) [57] and Google AppEngine (GAE) YAML-based specification language [16]. These abstractions are focused on a particular PaaS, thus they have no need to separate the platform from user requirements. Rather, they simply abstract away some details to make it easier for users to use the particular platform to deploy their apps. The abstractions only capture applications under the users' control and do not include platform service structures. As a result the abstractions *cannot* support compiling abstract specifications to different cloud platforms[2].

Systems like Maestro [22], Maestro-NG [23], Deis [9], and Flynn [15] are based on the Linux Containers [21] virtualization approach (specifically the Docker open-source engine [10]). Some of the description languages in these systems (specifically Maestro and MaestroNG) can capture dependencies among the containers (applications) through named channels. However, these specifications abstract *instances* (virtual machines), as opposed to the whole system. There is no formal model to define a globally consistent view of the system, and as a result once a system is deployed it is challenging to perform reliable configuration updates. Current Docker-based solutions are primarily focused on the initial configuration/deployment; maintenance is usually not addressed or they resort to a re-deployment process.

The lack of a consistent high-level abstraction describing the complete IT system creates a number of challenges in configuring cloud-based systems: network deployments and changes cannot be automatically validated, automated solutions are error-prone, incremental changes are challenging (if not impossible) to automate, and configuration definitions are unique to specific cloud providers and are not easily ported to other providers.

[2]Indeed, it appears that these abstractions will likely make it harder for users to move to other cloud providers as they are platform-specific.

### 2.2 Enabling Technologies

Several new technologies have facilitated the development of our current prototype. In particular, there have been several advancements in the configuration management tools (CMT) that help streamline the configuration management process. This is especially beneficial to our work, since those technologies are the perfect building blocks for our compilation process. To help the reader better understand our approach, we present a basic background on the state-of-the-art CMTs.

Two popular configuration management solutions are Chef [34] and Puppet [20]. We use Puppet but similar concepts exist in Chef as well. Puppet works by installing an agent on the host to be managed, which communicates with a controller (called the master) to receive configuration directives. Directives are written in a declarative language called Puppet *manifests*, which define the *desired configuration state* of the host (*e.g.*, installed packages, configuration files, running services, *etc*.). If the host's current state is different than the manifest received by the Puppet agent, the agent will issue appropriate commands to bring the system into the specified state.

In Puppet, manifests can be reused by extracting the directives and placing them in *classes*. Puppet classes use parameters to separate the configuration data (*e.g.*, IP addresses, port numbers, version numbers, *etc*.) from the configuration logic. Classes can be packaged together in a Puppet module for reuse. Typically, classes are bound to nodes in a master manifest known as the *site manifest*. Puppet can also be configured to use an external program such as External Node Classifier (ENC) [18] or Hiera [35] to provide specific configuration data to the classes that will be assigned to a node.

In the current prototype we use Hiera [35], which is a key/value look-up tool for configuration data. Hiera stores site-specific data and acts as a site-wide configuration file, thus separating the specific configuration information from the Puppet modules. Puppet classes can be populated with configuration data directly from Hiera, which makes it easier to re-use public Puppet modules "as is" by simply customizing the data in Hiera. Moreover, users can publish their own modules without worrying about exposing sensitive environment-specific data or clashing variable names. Hiera also supports module

```
class role::work_queue::default {

  $exports = hiera("exports")

  class { "profile::redis":
    port => $exports["redis"]["port"]
  }
}
```

Figure 1: Puppet Worker Queue Class

```
classes:
  - role::work_queue::default

exports:
  redis: { port: 6379 }
```

Figure 2: Hiera Configuration Data

customization by enabling the configuration of default data with multiple levels of overrides.

Figure 1 is an example of a Puppet class for a `worker_queue` based on Redis [47]. Puppet classes can be reused in different scenarios without hard-coding parameters: in this particular example there is only one parameter, `port`. The concrete value of this parameter (`$exports["redis"]["port"]`) is derived from Hiera (Figure 2), which is shown as `6379` but can be computed automatically by a program at runtime. This allows us to calculate parameters based on the up-to-date system model, as opposed to hardcoding them. We use this technology in the compilation process described later.

We should also emphasize that while our current prototype uses Puppet, ANCOR can work with many mature CMT solutions such as Chef, SaltStack [49], Bcfg2 [43], or CFEngine [44]. Two important properties are required for a CMT to be useable by ANCOR. First, the directives an agent receives dictates a *desired state* as opposed to commands for state changes, which allows configuration changes to be handled in the same way as the initial configuration. Second, there is a mechanism for reusable configuration modules (*e.g.*, Puppet classes) that become the building blocks, or the “instruction set,” into which ANCOR can compile the abstract requirement model. Depending on the specific CMT features, an orchestrator component might also be needed (especially in case the CMT employs only a pull-configuration model). An orchestrator component can be used on the CMT master node to trigger different actions on the CMT agents (achieve a push-configuration model).

## 3 The ANCOR Framework

Figure 3 shows the three major components of the ANCOR framework: the Operations Model, the Compiler, and the Conductor. The arrows denote information flow.

The key idea behind our approach is to abstract the functionality and structure of IT services into a model that is used to generate and manage concrete systems.

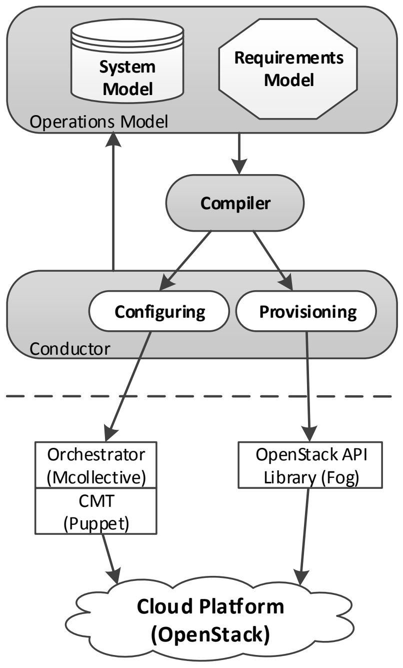


Figure 3: ANCOR Framework

We call this abstraction the *requirement model*. We also maintain the details of the concrete system in the *system model*. The two constitute the Operations Model. When ANCOR compiles a requirement model into a concrete, cloud-based system, the system model is populated with the details of the cloud instances and their correspondence to the requirement model. When the system changes, the system model is updated to ensure it has a consistent and accurate view of the deployment. Figure 14, part of the Appendix, shows the complete entity-relation diagram for the operations model.

The Compiler references the requirement model to make implementation decisions necessary to satisfy the abstract requirements and to instruct the conductor to orchestrate the provisioning and configuration of the instances. It can also instruct the conductor to perform user-requested configuration changes while ensuring the concrete system always satisfies the requirement model.

The Conductor consists of two sub-components, Provisioning and Configuring, which are responsible for interacting with the cloud-provider API, the CMT and orchestration tools (shown below the dashed line).

The ANCOR framework manages the relationships and dependencies between instances as well as instance clustering. Such management involves creating and deleting instances, adding/removing instances to/from clusters, and keeping dependent instances/clusters aware of configuration updates. The ANCOR framework simplifies network management as system dependencies are formalized and automatically maintained. Moreover, tra-

```
goals:
 ecommerce:
    name: eCommerce frontend
    roles:
      - weblb
      - webapp
      - worker
      - work_queue
      - db_master
      - db_slave

 roles:
  weblb:
    name: Web application load balancer
    min: 2
    is_public: true
    implementations:
      default:
        profile: role::weblb::default
    exports:
      http:{ type: single_port, protocol: tcp, number:80 }
    imports:
      webapp: http

  webapp:
    name: Web application
    min: 3
    implementations:
      default:
        profile: role::webapp::default
    exports:
      http: { type: single_port, protocol: tcp }
    imports:
      db_master: querying
      db_slave: querying
      work_queue: redis

 worker:
  name: Sidekiq worker application
  min: 2
  implementations:
    default:
      profile: role::worker::default
  imports:
    db_master: querying
    db_slave: querying
    work_queue: redis

 work_queue:
  name: Redis work queue
  implementations:
    default:
      profile: role::work_queue::default
  exports:
    redis: { type: single_port, protocol: tcp }

 db_master:
  name: MySQL master
  implementations:
    default:
      profile: role::db_master::default
  exports:
    querying: { type: single_port, protocol: tcp }

 db_slave:
  name: MySQL slave
  implementations:
    default:
      profile: role::db_slave::default
  min: 2
  exports:
    querying: { type: single_port, protocol: tcp }
  imports:
    db_master: querying
```

Figure 4: eCommerce Website Requirements Specification

ditional failures can also be addressed, thus increasing network resiliency. Next, we explain the requirement model, the way it is compiled into a concrete system, and how it is used to better manage that system.

## 3.1 Requirement Model

### 3.1.1 The ARML language

We specify the requirement model in a domain-specific language called the ANCOR Requirement Modeling Language (ARML). ARML's concrete syntax is based on YAML [12], which is a language that supports specification of arbitrary key-value pairs. The *abstract syntax* of ARML is described in the Appendix (Figure 13). Figure 4 shows an example ARML specification for an eCommerce website in Figure 5.

The example is a scalable and highly available eCommerce website on a cloud infrastructure, which adopts a multiple-layer architecture with the various clusters of services shown in Figure 5: web load balancer (Varnish [55]), web application (Ruby on Rails [48] ) with Unicorn [30], HAProxy [17] and Nginx [29]), database (MySQL [26]), worker application (Sidekiq [40]), and messaging queue (Redis [47]). Arrows indicate dependency between the clusters. Each cluster consists of mul-

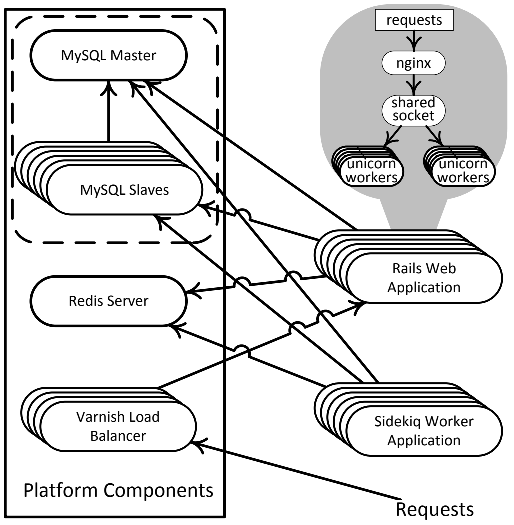


Figure 5: eCommerce Website

tiple instances that offer the same services. Clustering supports scaling via cluster expansion (adding more instances to the cluster) or contraction (removing instances from the cluster). The clustering strategies employed by these applications fall into two main categories: homogeneous and master-slave. In a homogeneous cluster all cluster members have the same configuration. If one of the instances stops working, another instance takes over. In master-slave, the master and slave instances have different configurations and perform different functions (*e.g.*, write versus read). If the master fails, a slave is promoted to be the master. In this example system, the web load balancer, web application, and the worker application employ the homogeneous clustering while the database employs master-slave (thus MySQL master and MySQL slaves form one cluster). Redis is used as a messaging queue. The clustering approach, mainly replication, supported by Redis is not suited for high-throughput queues.

A requirement model contains the specifications of system *goals* and *roles*. A *goal* is a high-level business goal (*e.g.*, blog website, eCommerce website, *etc.*) whose purpose is to organize the IT capabilities (*roles*) around business objectives. In Figure 4 there is a single system goal `ecommerce` that is supported by six roles.

A *role* defines a logical unit of configuration. Examples include a database role, a web application role, a message broker role, and so on. In essence, a role represents a group of similarly configured instances that provide the same functionality. In our model we use a single role to represent all the instances that achieve that functionality. For example, the web application instances in Figure 5 are configured identically (except for network addresses) and multiple load balancers dispatch incoming web requests to the instances in the web application cluster. We have a single role `webapp` for all the web application instances, and a `weblb` role for all the load balancer instances. The role-to-instance mapping is maintained in the system model.

A role may depend on other roles. A role uses a *channel* to interact with other roles. A channel is an interface *exported* (provided) by a role and possibly *imported* (consumed) by other roles. Channels could include a single network port or a range of ports. For instance, the `webapp` role exports an `http` channel, which is a TCP port (*e.g.,* 80). The `weblb` role imports the `http` channel from the `webapp` role. A role is a "black box" to other roles, and only the exported channels are visible interfaces. Using these interfaces the requirement model captures the dependencies between the roles.

The `webapp` role also imports three channels from various other roles: `querying` from `db_master`, `querying` from `db_slave`, and `redis` from `work_queue`. This means the `webapp` role depends upon three other roles: `db_master`, `db_slave`, and `work_queue`. The `min` field indicates the minimum number of instances that should be deployed to play the role. If `min` is not specified it's default value is 1. In the current prototype implementation the count of the instances will be equal to `min`. The requirement model addresses instance clustering naturally by requiring multiple instances to play a role. For homogeneous clusters this is easy to understand. For master-slave clusters, at least two roles are involved in the cluster, the master and the slave. The dependency information captured in the export/import relationship is sufficient to support calculating configuration changes when, for example, the master is removed from the cluster and a new node is promoted to be the master. So far we have not found any real-world clustering strategies that require explicitly modeling the cluster structure beyond the dependency relationship between the roles that form the cluster. If more general clustering strategies are needed, we will extend our requirement model to support this.

### 3.1.2 Role Implementation

Role names are system-specific and are chosen by the user or system engineers to convey a notion of the role's purpose in the system; there are no pre-defined role names in ARML. However, to automatically compile and maintain concrete systems, system engineers must provide the *semantics* of each role, which is specified in the role specification's *implementation* field. The implementation field defines how each instance must be configured to play the role. The conductor then uses this implementation information to properly configure and deploy the concrete instances. The value of the implementation field is thus dependent on the CMT being used. This process is similar to traditional programming language compilers where abstract code constructs are compiled down to machine code. The compiler must contain the semantics of each code construct in terms of machine instructions for a specific architecture. The analogy between our ANCOR compiler and a programming language compiler naturally begs the question: "what is the *architecture-equivalent* of a cloud-based IT system?" In other words, is there an interface to a "cloud runtime" into which we can compile an abstract specification?

It turns out that a well-defined interface between the requirement model and the "cloud runtime" is well within reach if we leverage existing CMT technologies. As explained in Section 2.2, there has been a general movement in CMT towards encapsulating commonly-used configuration directives into reusable, parameterized modules. Thus, one can use both community and custom modules to implement roles and populate those reusable knowledge units with parameters derived from our high-level requirement model. Potential role implementations must be specified in a role's "implementations" field (see Figure 4). A role may have multiple implementations since there could be more than one way to achieve its functionality. The compiler then selects an

```
class role::weblb::default {

  $exports = hiera("exports")
  $imports = hiera("imports")
---
  class { "profile::varnish":
    listen_port => $exports["http"]["port"] }

  $backends = $imports["webapp"]

  file { "default.vcl":
    ensure  => file,
    content =>
     template("role/weblb-varnish/default.vcl.erb"),
    path    => "/etc/varnish/default.vcl",
    owner   => root,
    group   => root,
    mode    => 644,
    require => Package["varnish"],
    notify  => Exec["reload-varnish"], }
}
```

Figure 6: Web Load Balancer Role Implementation

appropriate role implementation from those that satisfy all constraints levied by existing role implementations in the system. (The current prototype simply selects the first role implementation in the "implementations" field; we leave the constraint specification and implementation selection problems for future work.)

An important challenge was structuring the knowledge units so that they could be easily reused in different requirement models. Failing to have a proper role implementation design model would lead to rewriting every single role implementation from scratch. We adopted an approach similar to that used by Dunn [11]. We name role implementations based on their functionality and/or properties (*e.g.*, weblb) and use "profiles" to integrate individual components to embody a logical software stack. The software stack is constructed using community and custom modules as lower-level components[3].

For instance, in case of the load balancer: The `weblb` role is assigned the `default` implementation in the ARML specification (see Figure 4). Figure 6 is a Puppet class that shows the implementation that was defined as `default` for the `weblb`. Figure 7 pictures a sample of possible parameters that Puppet is getting through Hiera from the compiler for configuring one of the `weblb` instances. There are two parts in each role implementation (see Figure 6). The code before "`---`" imports operations model values from Hiera (*e.g.*, see Figure 7). The statements `hiera("exports")` and `hiera("imports")` query Hiera to find all the channels the web balancer will consume (*imports*) and the channels that it will make available to other roles (*exports*). These channels will be stored in two variables,

[3] All `default` role `implementations` used with ANCOR are available on GitHub: https://github.com/arguslab/ancor-puppet

```
{
 "exports": {
   "http": {
     "port": 80,
     "protocol": "tcp"
   }
 },
 "imports": {
   "webapp": {
     "webapp-ce66a264": {
       "ip_address": "10.118.117.16",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 42683,
         "protocol": "tcp"
       }
     },
     "webapp-84407edd": {
       "ip_address": "10.118.117.19",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 23311,
         "protocol": "tcp"
       }
     },
     "webapp-1ce1ce46": {
       "ip_address": "10.118.117.22",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 10894,
         "protocol": "tcp"
       }
     }
   }
 },
 "classes": [
   "role::weblb::default"
 ]
}
```

Figure 7: Specific `weblb` Paramaters Sample Exposed to Hiera by ANCOR

`"exports"` and `"imports"`. The web balancer will be instructed to expose an `http` channel on the particular port (in this case port 80, see "exports" in Figure 7), and will be configured to use all instances that are assigned to play the role `webapp`, from which it imports the `http` channel.

The default `weblb` implementation is based on the reusable Puppet "Varnish profile" (`profile::varnish` - see Figure 8). The `profile::varnish` Puppet class uses the necessary specified parameters to customize the Varnish installation. The parameters in `profile::varnish` (*e.g.*, `$listen_address`, `$listen_port`, *etc.*) are initialized with default values. These values will be overwritten in case they are specified in `role::weblb::default`. In the current example, `$listen_port` is the only parameter that will be overwritten (see Figure 6), the other parameters will keep their default values defined in

```
class profile::varnish(
  $listen_address = "0.0.0.0",

# $listen_port's default value "6081" will be
# OVERWRITTEN with the value passed
# from role::weblb::default
  $listen_port = 6081,

  $admin_listen_address = "127.0.0.1",
  $admin_listen_port = 6082) {

  apt::source { "varnish":
    location =>
      "http://repo.varnish-cache.org/ubuntu/",
    release => "precise",
    repos => "varnish-3.0",
    key => "C4DEFFEB",
    key_source =>
    "http://repo.varnish-cache.org/debian/GPG-key.txt",
  }
  package { "varnish":
    ensure => installed,
    require => Apt::Source["varnish"], }

  service { "varnish":
    ensure => running,
    require => Package["varnish"], }

  Exec {
   path => ["/bin", "/sbin", "/usr/bin", "/usr/sbin"]
  }

  exec { "reload-varnish":
    command => "service varnish reload",
    refreshonly => true,
    require => Package["varnish"] }

  file { "/etc/default/varnish":
    ensure => file,
    content =>
      template("profile/varnish/default.erb"),
    owner => root,
    group => root,
    mode => 644,
    notify  => Service["varnish"],
    require => Package["varnish"], }
}
```

Figure 8: Web Load Balancer Role Profile

```
# Configuration file for varnish
START=yes
NFILES=131072
MEMLOCK=82000
VARNISH_VCL_CONF=/etc/varnish/default.vcl
VARNISH_LISTEN_ADDRESS=<%= @listen_address %>
VARNISH_LISTEN_PORT=<%= @listen_port %>
VARNISH_ADMIN_LISTEN_ADDR=<%= @admin_listen_address %>
VARNISH_ADMIN_LISTEN_PORT=<%= @admin_listen_port %>
VARNISH_MIN_THREADS=1
VARNISH_MAX_THREADS=1000
. . .
```

Figure 9: Web Load Balancer, Varnish, Initialization Script: *default.erb* (used in *profile::varnish*)

```
<% @backends.each do |name, backend| %>
backend be_<%= name.sub("-", "_") %> {
  .host = "<%= backend["ip_address"] %>";
  .port = "<%= backend["http"]["port"] %>";

  .probe = {
    .url = "/";
    .interval = 5s;
    .timeout = 1s;
    .window = 5;
    .threshold = 3;
  }
}
<% end %>

director webapp round-robin {
  <% @backends.each_key do |name| %>
  {
    .backend = be_<%= name.sub("-", "_") %>;
  }
  <% end %>
}

sub vcl_recv {
  set req.backend = webapp;
}
```

Figure 10: Web Load Balancer, Varnish, Configuration File: *default.vcl.erb* (used in *role::weblb::default*)

profile::varnish. The parameters values (initialized in role::weblb::default or in profile::varnish) are passed to Figure 9 and Figure 10 to generate the customized Varnish configuration files, and this is all done by Puppet automatically at runtime.

Thus, a role implementation definition specifies *a concrete way to implement the intended functionality embodied by a role* by describing the invocation of predefined configuration modules with concrete parameters computed from the operations model. These role implementations are not only useful when generating the system, but also for modifying the system as it changes over time. For example, if a new instance is deployed to play the webapp role, the dependency structure in the operations model allows ANCOR to automatically find all the other roles that may be impacted (those depending on the webapp role) and use their role implementation to direct the CMT to reconfigure them so that they are consistent with the updated operations model.

ANCOR leverages existing CMT to define the role implementations, to minimize additional work that has to be done by the system engineers. For example, only information in Figure 6 is what one needs to write for ANCOR; Figure 7 is generated automatically by ANCOR; Figure 8, 9, and 10 are what one would have to specify anyway using Puppet.

## 3.2 The ANCOR Workflow

There are four main phases involved in creating and managing cloud-based systems using ANCOR.

1. Requirements model specification
2. Compilation choices specification
3. Compilation/Deployment
4. Maintenance

The first two phases result in the creation of the requirement model while the next two phases perform the actual deployment and maintenance of the cloud-based system.

### 3.2.1 Requirement Model Specification

In this phase, the user and system engineers work together to define the goals of the system, which may require significant input from various stakeholders. Next, they determine the roles required to achieve each goal and the dependencies among the roles. This task could be handled by the system engineers alone or in consultation with the user. The high-level requirement language ARML provides an abstract, common language for this communication.

### 3.2.2 Compilation Choices Specification

In this phase, system engineers define role semantics using pre-defined CMT modules. In our current prototype this is accomplished by defining the role implementations that invoke Puppet classes as described in Section 3.1.2. If no appropriate CMT modules exist, system engineers must define new profiles and place them into the repository for future use. In general, system engineers could specify multiple implementation choices for each role to provide the ANCOR compiler flexibility in choosing the appropriate implementation at runtime. One of the options available to system engineers is the specification of the desired operating system for each instance. Here again, different operating systems may be used for each implementation of a role. With a wide variety of choices available to systems engineers, a constraint language is needed to specify the compatibility among the various implementation choices; we leave this for future research.

### 3.2.3 Compilation/Deployment

Once the requirement model has been defined, the framework can automatically compile the requirements into a working system on the cloud provider's infrastructure. This process has seven key steps:

1. The framework signals the compiler to deploy a specific requirement model.
2. The compiler makes several implementation decisions including the number of instances used for each role and the flavors, operating systems, and role implementations used.
3. The compiler signals the conductor component to begin deployment.
4. The conductor interacts with the OpenStack API to provision instances and create the necessary security rules (configure the cloud's internal firewall). The provisioning module uses a package such as *cloud-init* to initialize each cloud instance, including installing the CMT and orchestration tool agents (*e.g.*, the Puppet agent and MCollective [19] agent).
5. Once an instance is live, the message orchestrator (*e.g.*, MCollective) prepares the instance for configuration by distributing its authentication key to the CMT master (*e.g.*, Puppet master).
6. The configuration is pushed to the authenticated instances using the CMT agent and, if needed, the orchestrator (*e.g.*, Puppet agent and MCollective).
7. System engineers may check deployed services by using system monitoring applications such as Sensu [50] or Opsview [46] , or by directly accessing the instances.

In step 6, configuration is carried out via the Hiera component while configuration directives (node manifests) are computed on the fly using ANCOR's operations model. This ensures that the parameters used to instantiate the Puppet modules always reflect the up-to-date system dependency information.

### 3.2.4 Maintenance

Once the system is deployed in the cloud, system engineers can modify the system. If the change does not affect the high-level requirement model, the maintenance is straightforward. The compiler will track the impacted instances using the operations model and reconfigure them using the up-to-date system information. A good example for this type of change is cluster expansion/contraction.

**Cluster expansion** is used to increase the number of instances in a cluster (*e.g.*, to serve more requests or for high-availability purposes).

1. System engineers instruct the compiler to add instances to a specific role. In future work we also would like to allow monitoring modules to make expansion decisions as well.
2. The compiler triggers the conductor component to create new instances, which automatically updates the ANCOR system model.

3. The compiler calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.

**Cluster contraction** is the opposite of cluster expansion. The main goal of cluster contraction is to reduce the number of instances in a cluster (*e.g.*, to lower cost).

1. System engineers instruct the compiler to mark a portion of a role's instances for removal.
2. The compiler calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.
3. The compiler triggers the conductor component to remove the marked instances.

If the change involves major modifications in the requirement model (*e.g.,* adding/removing a role), ANCOR will need to re-compile the requirement model. How to perform "incremental recompilation" that involve major structural changes without undue disruption will be a topic for future research.

## 4 Prototype Implementation

We built a prototype (Figure 3) in Ruby (using Rails, Sidekiq and Redis) to implement the ANCOR framework using OpenStack as the target cloud platform. The operations model is stored in MongoDB collections using Rails. ANCOR employs straight-forward type-checking to ensure that the requirement model is well-formed (*e.g.*, allowing a role to import a channel from another role only if the channel is exported by that role). The compiler references the MongoDB document collections that store the operations model and interacts with the conductor using a Redis messaging queue and Sidekiq, a worker application used for background processing. The conductor interacts with the OpenStack API through Fog (a cloud services library for Ruby) to provision the network, subnets and instances indicated by the compiler. Once an instance is live, the configuration module uses Puppet and MCollective to configure it using the manifest computed on the fly based on the operations model. The conductor also interacts with the system model and updates the provided system model database every time it performs a task (provisioning or configuration). Therefore, the system model stored in the MongoDB datastore will always have an updated picture of the system. Obviously, the different role implementation choices (*e.g.,* Sidekiq, Redis or Rails) used to build the eCommerce website example scenario (Figure 5) are independent from the components that leverage Sidekiq, Redis and Ruby on Rails in the ANCOR framework prototype (Figure 3)

In the current implementation we are using a workflow model that is based on chained and parallel tasks processing. Once the ARML specification is entered by the user, the specification will be parsed and the requirement model will be encountered. Next, the compiler steps in and based on the requirement model it chooses the number of instances that play a role, the role implementations, the IP addresses, the channels (port number and/or sockets that will be consumed or exposed), *etc.* Then the compiler populates the system model and creates various tasks that it passes to the worker queue. A task can be viewed as an assignment that is passed to a background (worker) process. In ANCOR, Sidekiq is used for background processing. Tasks are stored in the database and have several attributes (*e.g.*, type, arguments, state, context) A task can be related to provisioning (*e.g.*, using Fog) or to configuring an instance (*e.g.*, push configuration from Puppet master to Puppet agent). In case other tasks (*e.g.*, deploy instance) depend on the execution of the current task (*e.g.*, create network) a *wait handle* is created. Wait handles can be viewed as the mechanism used by the tasks to signal dependent tasks when they finished execution. A task creates a wait handle object that stores the ids of the tasks that wait for it to execute. Once the task finished the wait handle triggers all the dependent tasks to execute. The purpose of a wait handle is to start, resume or suspend the dependent tasks. Using this approach we can resume or suspend a task several times including tasks related to the orchestration tool (MCollective) and the CMT (Puppet). Independent tasks (*e.g.*, two different *deploy instance* tasks) will be executed in parallel employing locks on certain shared resources. The ANCOR prototype code, detailed instructions on how to deploy/run it and a detailed document containing specific implementation details are available online at https://github.com/arguslab/ancor.

### 4.1 Using ANCOR

The current framework implementation exposes a REST API [14] to it's clients. The current clients include a Command-Line Interface (CLI), a web-browser dashboard and also the Puppet master (specifically the Hiera module). Through the REST API, Hiera is obtaining the specific configuration details (*e.g.*, imported and exported channels - `$exports` and `$imports` arrays, see Figure 1) from the compiler in order to customize the Puppet modules that are part of the chosen role implementation (*e.g.*, see Figure 2). The CLI and the dashboard are used to deploy, manage, visualize (in case of the dashboard) and delete ANCOR deployments.

One can use the CLI to deploy, manage and delete the eCommerce website example using several key commands:

1. `ancor environment plan eCommerce.yaml` – plans the deployment (`eCommerce.yaml` is shown in Figure 4)
2. `ancor role list` – lists the current set of roles
3. `ancor instance list` – lists the current set of instances
4. `ancor environment commit` – deploys the environment on the cloud infrastructure
5. `ancor task list` – displays the current progress of the deployment
6. `ancor instance add webapp` – used to add another instance for the `webapp` role after all `tasks` are `completed`
7. `ancor environment list` – used to unlock the environment after all tasks are `completed`
8. `ancor environment remove production` – deletes the current deployment

More options and instructions on using the ANCOR CLI and the dashboard are available on the framework website.

## 4.2 Prototype Evaluation

The objective of our evaluation is to measure the dynamics of change management. This implies measuring how much the performance of a deployed IT system is influenced when adding and removing instances using ANCOR. Throughput and latency values are highly dependent on the applications' configurations and on the underlying cloud infrastructure. Thus, we are not focused on the throughput and latency themselves but on the *difference* between the baseline measurements and the measurements during system changes. We evaluated our prototype using two IT system setups: a basic blogging website (Figure 11) and the eCommerce website scenario (Figure 5). These scenarios are also available online. As we add other scenarios into the repository and integrate ANCOR with different cloud infrastructures (AWS will likely be the next infrastructure supported), we plan to expand this evaluation.

The current testing scenarios were deployed on a private cloud testbed in the Argus CyberSecurity Laboratory at Kansas State University. The cloud testbed consists of fifteen Dell PowerEdge R620 servers, each of which is equipped with 2 x Intel Xeon E-2660 v2 Processor@2.20GHz, 128GB of RAM, 4 x 1TB 7.2K HDD running in RAID0 (Stripe) and an Intel dual port 10GbE DA/SFP+ network card. Moreover, each server is connected to the main switch (Dell Force10 - S4810P switch) using two bounded 10GbE ports.

We deployed an OpenStack Havana infrastructure on these machines using Mirantis' open-source framework Fuel [24]. The infrastructure consists of one controller node and fourteen compute nodes.

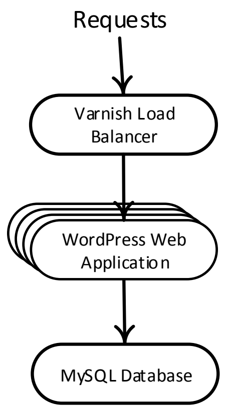


Figure 11: Blogging Website

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/- Stdev |
| Latency | 1.03ms | 1.28ms | 305.05ms | 99.73% |
| 5999.68 requests/sec, 1800296 requests in 5 min | | | | |
| Timeouts: 5052 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 1: Benchmarking - Blogging Website Scenario (baseline) with Caching

*wrk* [58], an HTTP benchmarking tool, was used to test the system's availability and performance while managing various clusters (*i.e.,* adding/removing instances to/from a cluster). We ran the benchmarking tool on the initially deployed scenarios and established a baseline for every component in our measurements. *wrk* was launched from client machines that are able to access the websites (*i.e.,* connect to the load balancer). We targeted various links that ran operations we implemented to specifically test the various system components (*e.g.,* read/write from/to the database ). After establishing the baseline, we started adding and removing instances to and from different clusters while targeting operations that involve the deepest cluster in the stack (database slave for the eCommerce setup). In case of the blogging website scenario we focused on the web application. The performance in the three cases are very close since we only added and removed one instance in the experiments. All caching features at the application level were disabled in both scenarios. Having caching features enabled would have prevented us from consistently reaching the targeted components; however the performance would be *greatly* improved. For instance, Table 1 exposes the baseline results for accessing a WordPress [56] posts from the blogging website setup (Figure 12) with caching enabled.

```
goals:
  wordpress:
    name: Wordpress blog
    roles:
      - db
      - webapp
      - weblb

roles:
  weblb:
    name: Web application load balance
    is_public: true
    implementations:
      default:
        profile: role::weblb::default
    exports:
      http: { type: single_port, protocol: tcp, number: 80 }
    imports:
      webapp: http

  webapp:
    name: Web application
    min: 2
    implementations:
      default:
        profile: role::weblb::default
    exports:
      http: { type: single_port, protocol: tcp }
    imports:
      db: querying

  db:
    name: MySQL database
    implementations:
      default:
        profile: role::weblb::default
    exports:
      querying: { type: single_port, protocol: tcp }
```

Figure 12: Blogging Website Requirements Model

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/- Stdev |
| Latency | 458.89ms | 292.65ms | 1.30s | 73.86% |
| 86.18 requests/sec, 25855 requests in 5 min | | | | |
| Timeouts: 0 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 2: Benchmarking - Blogging Website Scenario - `webapp` cluster (baseline)

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/- Stdev |
| Latency | 418.76ms | 268.63ms | 1.88s | 78.65% |
| 96.63 requests/sec, 28989 requests in 5 min | | | | |
| Timeouts: 0 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 3: Benchmarking - Blogging Website Scenario - `webapp` cluster (adding one instance)

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/- Stdev |
| Latency | 456.13ms | 325.94ms | 1.34s | 73.53% |
| 89.49 requests/sec, 26849 requests in 5 min | | | | |
| Timeouts: 0 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 4: Benchmarking - Blogging Website Scenario - `webapp` cluster (removing an instance)

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/- Stdev |
| Latency | 16.75ms | 18.92ms | 333.76ms | 91.12% |
| 601.21 requests/sec, 180412 requests in 5 min | | | | |
| Timeouts: 4728 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 5: Benchmarking - `db_slave` cluster (baseline)

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/-Stdev |
| Latency | 17.21ms | 20.22ms | 161.34ms | 93.13% |
| 771.71 requests/sec, 231563 requests in 5 min | | | | |
| Timeouts: 4292 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 6: Benchmarking - `db_slave` cluster (adding one instance)

### 4.2.1 Blogging Website Scenario

The basic blogging website is pictured in Figure 11. When adding instances to the WordPress web application cluster, latency and throughput improve (Table 3), which is as expected. When instances are removed from the cluster the performance is slightly worse than the add-instance case, due to the decreased number of instances in the cluster. But it is still slightly better than the baseline, which has the less number of instances.

### 4.2.2 eCommerce Website Scenario

When adding instances to the MySQL slaves cluster, latency slightly increases but throughput is improved (Table 6). When instances are removed from the cluster both performance metrics were negatively affected (Table 7).

These results show that 1) the ANCOR system is reliable in generating a concrete IT system in the cloud from a high-level requirement model specification; and 2) the formal models in ANCOR help facilitate orchestrating changes in the cloud-based IT system such as scaling.

It is important not to disregard the fact that all benchmarking results were influenced by the way applications were configured (*e.g.*, disabled caching functionality and "hot-swap" feature). The "hot-swap" feature loads an updated configuration without restarting a service (*e.g.*, Unicorn), which could further improve performance if one further tunes these applications.

| **4** threads and **40** connections | | | | |
|---|---|---|---|---|
| | Avg | Stdev | Max | +/-Stdev |
| Latency | 15.91ms | 23.18ms | 417.68s | 93.95% |
| 500.59 requests/sec, 150202 requests in 5 min | | | | |
| Timeouts: 4903 | | | | |
| Errors (non-2xx or 3xx HTTP responses): 0 | | | | |

Table 7: Benchmarking - `db_slave` cluster (removing one instance)

## 5 Discussion and Future Work

Our requirements specification approach and the implemented ANCOR framework offer system engineers the same flexibility as in a typical IaaS model. This means that engineers can keep their workflow using their preferred configuration management tools (*e.g.,* Puppet and Chef) and orchestration tools (*e.g.,* MCollective). They have the option to do everything in their preferred ways up to the point where they connect the components (services) together. For example, system engineers have the option of using predefined configuration modules and of leveraging the contributions from the CMT community. Or they can write their own manifests or class definitions to customize the system in their own ways. ANCOR can leverage all of these and does not force the system engineers to use particular low-level tools or languages; rather it provides the ability to manage the whole system based on a high-level abstraction.

The high-level requirement model we developed could also facilitate tasks like failure diagnosis and system analysis to identify design weaknesses such as single point of failures or performance bottlenecks. The system dependency information specified in the requirement model and maintained in the operations model allows for more reliable and streamlined system updates such as service patching. It also allows for more fine-grained firewall setup (*i.e.*, only allows network access that is consistent with the system dependency), and enables porting systems to different cloud providers in a more organized manner (*e.g.*, one can take the up-to-date requirement model and compile it to a different cloud provider's infrastructure, and then synchronize data from the old one to the new one).

One of our future work directions is to construct a proactive change mechanism as part of the compiler. This mechanism would randomly select specific aspects of the configuration to change (*e.g.*, replacing a portion of the instances with freshly created ones with different IP addresses, application ports, software versions, *etc.*); the changes would be automatically carried out by the conductor component. Moreover, changes can be scheduled based on security monitoring tools (*e.g.*, Snort [51], Suricata [52], SnIPS [59], *etc.*) or on component behavior and interaction reports (using an approach similar to [1]). Our goal in this is to analyze the possible security benefits as well as to measure and analyze the performance, robustness, and resilience of the system under such disturbance. We are also considering adding SLA (Service Level Agreement) related primitives to the requirement model. Furthermore, we are looking into adding other CMT modules to support the use of multiple configuration management tools such as Chef. In addition, we are planning on developing built-in Docker support modules and switching to the OpenStack AWS compatible API that will enable us to use the same provider module on different cloud platforms.

## 6 Related Work

There has been a general trend towards creating more abstractions at various levels of cloud service offerings. Some of the solutions even use similar terminologies and features as those in ANCOR. For example, some solutions also use the term "role" in a similar way to ours [11, 57], and others have adopted named channels to describe dependencies in configuration files [22, 23, 57]. Thus it is important to describe the fundamental differences between the abstraction used in ANCOR and those in the other solutions, so that one does not get confused by the superficial similarities between them.

Abstractions used in the various PaaS solutions such as the Windows Azure service definition schema [57] and Google AppEngine YAML-based specifications [16] allow users to define their cloud-based applications. These abstractions, while useful for helping users to use the specific PaaS platform more easily, do not serve the same purpose as ARML. In particular, they only define user-provided applications and not the complete IT system in the cloud, since the important platform components are not modeled. Thus these abstractions cannot be used to compile into different implementation choices or platforms. They are tied to the specific PaaS platform and thus will never separate the user requirements from the platform details. Using these abstractions will lock the users in to the specific PaaS vendor, while ANCOR will give users complete flexibility as to implementation choices at all levels, including platforms.

Docker container-based solutions such as Maestro [22], Maestro-NG [23], Deis [9], and Flynn [15] provide management aid for deploying cloud instances using the Linux Containers virtualization approach. Some of them (Maestro and Maestro-NG) also have environment descriptions (in YAML) for the Docker instances that include named channels to capture dependencies. These solutions can automate initial deployment of cloud instances and make it easier to stand up a PaaS, but again they take a different approach and do not provide the same level of abstraction that supports the vision outlined in Section 1. Specifically, the abstractions provided by their environment descriptions are focused on instances as opposed to the complete IT system, the container is the unit of configuration, and maintenance tasks are done by progressing through containers. It is not clear how much the container-based solutions can help alleviate the long-term maintenance problem of cloud-based IT systems. Moreover, the container-based solutions are tied to Linux environments and Docker, which is still under heavy development and not ready for production use. We also summarized a few other features to differentiate ANCOR from the other solutions in Table 8. ANCOR can be used to deploy systems using other orchestration tools such as Flynn and Deis in conjunction with traditional IT systems. As shown in Table 8, ANCOR is currently the most general and flexible management solution available.

Several companies are developing cloud migration technologies. While some appear to internally use abstractions to support migration [8, 13], no details are available for independent evaluation. Our approach is more fundamental in the sense that we *build* systems using the abstraction and, smoother and more reliable migration could be a future product of our approach. Rather than creating technology specifically to replicate existing systems, we aim to fundamentally change the way cloud-based systems are built and managed, which includes enabling dynamic and adaptive system changes, reducing human errors, and supporting more holistic security control and analysis.

Often, solutions like AWS CloudFormation [4], OpenStack Heat [31] or Terraform [53] may be, mistakenly, viewed as being at the same level of abstraction with ANCOR. These solutions are primarily focused on building and managing the infrastructure (cloud resources) by allowing the details of an infrastructure to be captured into a configuration file. CloudFormation and Heat manage AWS/OpenStack resources using templates (*e.g.*, Wordpress template [33], MySQL template [32], *etc.*), and they do not separate user requirements from system implementation details. The templates have the potential to integrate well with configuration management tools but there is no model of the structure and dependencies of the system. Thus, it cannot achieve one main objective of ANCOR which is to use the operations model to maintain the system, *e.g.*, updating dependencies automatically while replacing instances. Terraform similarly uses configuration files to describe the infrastructure setup, but it goes even further by being cloud-agnostic and by enabling multiple providers and services to be combined and composed [54].

Juju [41] is a system for managing services and works at a similar level as ANCOR. It resides above the CMT technologies and has a way of capturing the dependencies between software applications (services). It can also interact with a wide choice of cloud services or bare metal servers. The Juju client works on multiple operating systems (Ubuntu, OS X, and Windows) but Juju-managed services run primarily on Ubuntu servers, although support for CentOS and a number of Windows-based systems will be available in the near future [42]. While we were aware of the existence of Juju at the time of this paper's writing, the lack of formal documentation on how Juju actually works, the services running only on Ubuntu, and the major changes in the Juju project (*e.g.*, code base was completely rewritten in Go) kept us away from this project. We recently reevaluated Juju and discovered fundamental similarities between ANCOR and Juju. Even so, there are subtle differences that make the two approaches work better in different environments. For instance, the ANCOR approach adopts a more "centralized" management scheme in terms of deciding the configuration parameters of dependent services, while Juju adopts a negotiation scheme between dependent services (called relations in Juju) to reach a consistent configuration state across those services. Depending on the need for change in the system, the ANCOR approach may be more advantageous when it comes to a highly dynamic system with proactive changing (*e.g.*, in a moving target defense system).

Our approach has benefited from the recent development in the CMT technologies that have provided the building blocks (or "instruction sets") for our compiler. The general good practice in defining reusable configuration modules such as those advocated by Dunn [11] is aligned very well with the way we structure the requirement model. Thus our approach can be easily integrated with those CMT technologies.

Sapuntzakis *et al.* [37] proposed the configuration language CVL and the Collective system to support the creation, publication, execution, and update of virtual appliances. CVL allows for defining a network with multiple appliances and passing configuration parameters to each appliance instance through key-value pairs. The decade since the paper was published has seen dramatic improvement in configuration management tools such as Puppet [20] and Chef [34], which has taken care of specifying/managing the configuration of individual machines. Our work leverages these mature CMTs and uses an abstraction on a higher level. In particular, ARML specifies the dependency among roles through explicit "import" and "export" statements with the channel parameters, which are translated automatically to concrete protocol and port numbers by the integration of operations model and the CMT. While CVL does specify dependency among appliances through the "provides" and "requires" variables, they are string identifiers and not tied to configuration variables of the relevant appliances (*e.g.*, the "DNS host" configuration parameter of an LDAP server). In the CVL specification of the virtual appliance network, the programmer would need to take care in passing the correct configuration parameters (consistent with the dependency) to the relevant appliances. In ANCOR this is done automatically by the coordination between the CMT and the operations model (compiled from the high-level ARML specification). This also allows for easy adaptation of the system (*e.g.*, cluster expansion and contraction).

Begnum [6] proposed MLN (Manage Large Networks) that uses a light-weight language to describe a

| Offering | Focus | Platform | Multiple Cloud Infrastructures | CMTs or other similar structures |
|---|---|---|---|---|
| OpenShift | Private PaaS | RHEL | Yes | None |
| Flynn | Private PaaS | Linux | Yes | Heroku Buildpacks, Docker |
| Deis | Private PaaS | Linux | Yes | Heroku Buildpacks, Chef, Docker |
| OpsWorks AWS | General | Ubuntu | No | Chef |
| Maestro | Single-host | Linux | Yes | Docker (based on LXC) |
| Maestro-NG | General | Linux | Yes | Docker (based on LXC) |
| Google AppEngine | PaaS | Linux | No | None |
| Heroku | PaaS | Linux | No | Heroku Buildpacks |
| Windows Azure | PaaS | Windows | No | None |
| Ubuntu Juju | General | Ubuntu, CentOS, Windows | Yes | Charms |
| ANCOR | General | Any | Yes | Puppet |

Table 8: Current Solutions Comparison

virtual machine network. Like ANCOR, MLN uses off-the-shelf configuration management solutions instead of reinventing the wheel. A major difference between ANCOR and MLN is that ANCOR captures the instance dependency in the requirement model, which facilitates automating configuration of a complete IT system and its dynamic adaptation. ANCOR achieves this by compiling the abstract specification to the operations model, which is integrated with the CMT used to deploy/manage the instances.

Plush [2] is an application management infrastructure that provides a set of abstractions for specifying, deploying, and monitoring distributed applications (*e.g.,* peer-to-peer services, web search engines, social sites, *etc*.). While Plush's architecture is flexible, it is not targeted towards cloud-based enterprise systems and it is unclear whether system dependencies can be specified and maintained throughout the system lifecycle.

Narain pioneered the idea of using high-level specifications for network infrastructure configuration management in the ConfigAssure [27, 28] project. ConfigAssure takes formally specified network configuration constraints and automatically finds acceptable concrete configuration parameter values using a Boolean satisfiability (SAT) solver. While we adopt Narain's philosophy of using formal models to facilitate system management, the cloud problem domain is different from that of network infrastructure, thus requiring new models and methodologies.

Use of higher-level abstractions to improve system management has also been investigated in the context of Software-Defined Networking (SDN). Monsanto *et al.* introduced abstractions for building applications from independent modules that jointly manage network traffic [25]. Their Pyretic language and system supports specification of abstract network policies, policy composition, and execution on abstract network topologies. Our ANCOR language and system adopts a similar philosophy for cloud-based deployment and management.

## 7 Conclusion

Separating user requirements from the implementation details has the potential of changing the way IT systems are deployed and managed in the cloud. To capture user requirements, we developed a high-level abstraction called the requirement model for defining cloud-based IT systems. Once users define their desired system in the specification, it is automatically compiled into a concrete cloud-based system that meets the specified user requirements. We demonstrate the practicality of this approach in the ANCOR framework. Preliminary benchmarking results show that ANCOR can improve manageability and maintainability of a cloud-based system and enable dynamic configuration changes of a deployed system with negligible performance overhead.

## 8 Acknowledgments

This work was supported by the Air Force Office of Scientific Research award FA9550-12-1-0106 and U.S. National Science Foundation awards 1038366 and 1018703. Any opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the above agencies.

## A Appendix

The *abstract syntax* of ARML is shown in Figure 13. We use $A \Rightarrow B$ to represent the key-value pairs (written as "$A : B$." in YAML). Upper-case identifiers represent non-terminal symbols, lower-case identifiers represent terminal symbols, and ARML keywords are distinguished by bold font.

Figure 14 shows the complete entity-relation diagram for the operations model. The arrows indicate the direction of references in the implementation (one-way or two-way references) and the associated multiplicity (1-to-1, 1-to-n, or n-to-n ). For instance, one role may support multiple goals, and multiple roles could support one goal. Thus the multiplicity between goal and role is n-to-n.

*ReqModel* ::= **goals** ⇒ *GoalSpec*$^+$
**roles** ⇒ *RoleSpec*$^+$

*GoalSpec* ::= goalID ⇒
[**name** ⇒ string]
**roles** ⇒ roleID$^+$

*RoleSpec* ::= roleID ⇒
[**name** ⇒ string]
[**min** ⇒ integer]
[**exports** ⇒ *ChannelSpec*$^+$]
[**imports** ⇒ *ImportSpec*$^+$]
**implementations** ⇒ *ImplementationSpec*$^+$

*ChannelSpec* ::= chanelID ⇒
(**type** ⇒ channelTypeID, *ChannelAttr*$^*$)

*ChannelAttr* ::= attributeID ⇒ value

*ImportSpec* ::= roleID ⇒ channelID$^+$

*ImplementationSpec* ::= implementationID ⇒ value

*goalID, roleID, channelID, attributeID, channelTypeID, strategyID, implementationID, clusterID, tag* are symbols. *integer* and *string* are defined in the usual way.

Figure 13: ARML Grammar

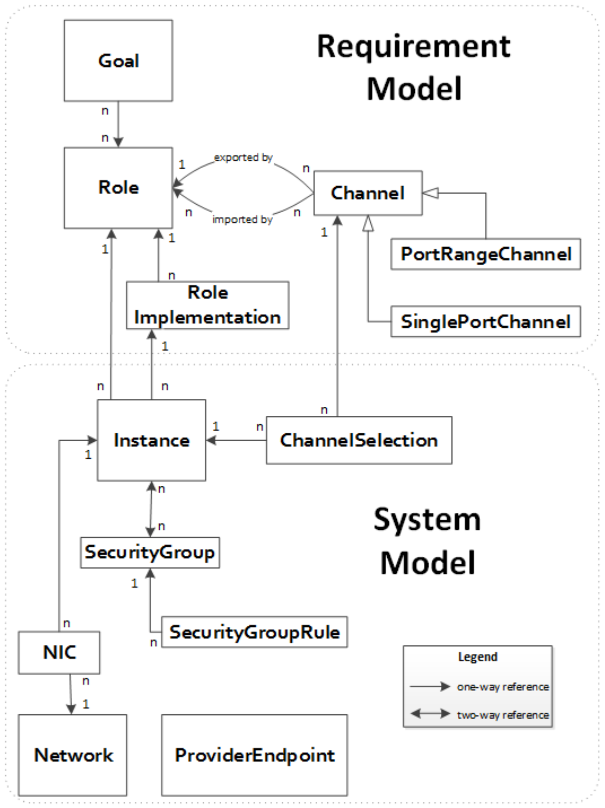


Figure 14: Operations Model

The system model is a local reflection of the cloud-based IT system and, as previously mentioned, it is used for bookkeeping. This enables the user to track instances in terms of roles. Furthermore, the system model bridges the gap between the more abstract requirement model and the many different concrete systems that can implement it. The requirement model is considered read-only by the rest of the framework. On the other hand, the system model can be updated by every component in the framework.

An *instance* is a virtual machine that is assigned to play a role. A role can be played by more than one instance but an instance currently can play only one role. A role can have one or more ways of being implemented. This aspect is captured in *RoleImplementation*. *RoleImplementation* is equivalent to *Scenario* in the current prototype code. A *NIC* stores the MAC address(es) that belong to an instance and a *Network* stores the network(s) that an instance is connected to. Moreover, an instance has access to the ports that a role consumes or exposes (channels) through *ChannelSelection*. A *Channel* can be a single port or a range of ports. The cloud provider firewall configuration (known as "security groups" in OpenStack) is captured in *SecurityGroup*. One *SecurityGroup* can have multiple configuration entries, *SecurityRules*. *ProviderEndpoint* captures the cloud platform specific API. This component makes it easier to integrate ANCOR with different cloud providers.

## References

[1] A.J.Oliner and A.Aiken. A query language for understanding component interactions in production systems. *In Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, pages 201–210, 2010.

[2] J. Albrecht, C. Tuttle, R. Braud, D. Dao, N. Topilski, A.C. Snoeren, and A. Vahdat. Distributed Application Configuration, Management, and Visualization with Plush. *In ACM Transactions on Internet Technology (Volume: 11, Issue: 2, Article No. 6)*, 2011.

[3] M. Armbust, A. Fox, R Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, pages 50–58, 2010.

[4] AWS. CloudFormation - accessed 7/2014. https://aws.amazon.com/cloudformation/.

[5] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A Security Analysis of Amazon's Elastic Compute Cloud Service. *In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1427–1434, 2012.

[6] K.M. Begnum. Managing Large Networks of Virtual Machines. *In Proceedings of the 20th USENIX conference on Large Installation System Administration (LISA)*, 2006.

[7] S. Bugiel, S. Nürnberger, T. Pöppelman, A. Sadeghi, and T. Schneider. AmazonIA: When Elasticity Snaps Back. *In Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, pages 389–400, 2011.

[8] CloudVelocity. One hybrid cloud software. Technical report, CloudVelocity Software, 2013.

[9] Deis website - accessed 3/2014. http://deis.io/overview/.

[10] Docker. Learn What Docker Is All About - accessed 1/2014. https://www.docker.io/learn_more/.

[11] Craig Dunn. Designing Puppet: Roles and Profiles - accessed 10/2013. http://www.craigdunn.org/2012/05/239/.

[12] C. C. Evans. YAML - accessed 8/2013. http://yaml.org/.

[13] William Fellows. 'Run any app on any cloud' is CliQr's bold claim. 451 Research, Jan 2013. http://cdn1.hubspot.com/hub/194983/Run_any_app_on_any_cloud.pdf.

[14] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[15] Flynn website - accessed 4/2014. https://flynn.io/.

[16] Google Developers - accessed 12/2013. https://developers.google.com/appengine/docs/python/config/appconfig.

[17] HAProxy website - accessed 2/2014. http://haproxy.1wt.eu/.

[18] Puppet Labs. External Node Classifiers - accessed 9/2013. http://docs.puppetlabs.com/guides/external_nodes.html.

[19] Puppet Labs. What is MCollective and what does it allow you to do? - accessed 9/2013. http://puppetlabs.com/mcollective.

[20] Puppet Labs. What is Puppet? - accessed 9/2013. http://puppetlabs.com/puppet/what-is-puppet.

[21] LXC - Linux Containers - accessed 1/2014. http://linuxcontainers.org/.

[22] Maestro Github repository - accessed 1/2014. https://github.com/toscanini/maestro.

[23] Maestro-NG Github repository - accessed 4/2014. https://github.com/signalfuse/maestro-ng.

[24] Mirantis Software Website - accessed 3/2014. http://software.mirantis.com/.

[25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. *In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2013.

[26] MySQL website - accessed 9/2013. http://www.mysql.com/.

[27] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[28] Sanjai Narain, Sharad Malik, and Ehab Al-Shaer. Towards eliminating configuration errors in cyber infrastructure. In *Proceedings of 4th IEEE Symposium on Configuration Analytics and Automation (SafeConfig)*, pages 1–2. IEEE, 2011.

[29] Nginx website - accessed 1/2014. http://nginx.org/.

[30] Unicorn! Repository on GitHub. What it is? - accessed 4/2014. https://github.com/defunkt/unicorn.

[31] OpenStack. Heat - accessed 7/2014. https://wiki.openstack.org/wiki/Heat.

[32] OpenStack. Heat/CloudFormation MySQL Template - accessed 5/2014. https://github.com/openstack/heat-templates/blob/master/cfn/F17/MySQL_Single_Instance.template.

[33] OpenStack. Heat/CloudFormation Wordpress Template - accessed 5/2014. https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_With_LB.template.

[34] Opscode. Chef - accessed 3/2014. http://www.opscode.com/chef/.

[35] PuppetLabs Docs - accessed 2/2014. http://docs.puppetlabs.com/hiera/1/.

[36] RightScale. White Paper - Quantifying the Benefits of the RightScale Cloud Management Platform - accessed 8/2013. http://www.rightscale.com/info_center/white-papers/RightScale-Quantifying-The-Benefits.pdf.

[37] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum. Virtual Appliances for Deploying and Maintaining Software. *In Proceedings of the 17th USENIX conference on Large Installation System Administration (LISA)*, pages 181–194, 2003.

[38] Service-now.com. White Paper - Managing the Cloud as an Incremental Step Forward - accessed 8/2013. http://www.techrepublic.com/resource-library/whitepapers/managing-the-cloud-as-an-incremental-step-forward/.

[39] Amazon Web Services. AWS OpsWorks - accessed 4/2014. http://aws.amazon.com/opsworks/.

[40] Sidekiq Repository on GitHub - accessed 2/2014. https://github.com/mperham/sidekiq.

[41] Ubuntu. Juju - accessed 7/2014. https://juju.ubuntu.com/.

[42] Steven J. Vaughan-Nichols. Canonical Juju DevOps Tool Coming to CentOS and Windows. http://www.zdnet.com/canonical-juju-devops-tool-coming-to-centos-and-windows-7000029418/.

[43] BCFG2 Website. What is Bcfg2? - accessed 4/2014. http://docs.bcfg2.org/.

[44] CFEngine Website. What is CFEngine? - accessed 3/2014. http://cfengine.com/what-is-cfengine.

[45] OpenStack Website. Open Source Software for Building Private and Public Clouds - accessed 4/2014. http://www.openstack.org/.

[46] OpsView Website. OpsView - accessed 4/2014. http://www.opsview.com/.

[47] Redis Website. Redis - accessed 4/2014. http://redis.io/.

[48] Ruby Website. Ruby is... - accessed 4/2014. https://www.ruby-lang.org/en/.

[49] SaltStack Website. What is SaltStack? - accessed 4/2014. http://saltstack.com/community.html.

[50] Sensu Website. Sensu - accessed 4/2014. http://sensuapp.org/.

[51] Snort Website. What is Snort? - accessed 4/2014. http://www.snort.org/.

[52] Suricata Website. - accessed 4/2014. http://suricata-ids.org/.

[53] Terraform Website. Terraform - accessed 9/2014. http://www.terraform.io/intro/index.html.

[54] Terraform Website. Terraform vs. Other Software - accessed 9/2014. http://www.terraform.io/intro/vs/cloudformation.html.

[55] Varnish Cache Website. About - accessed 4/2014. https://www.varnish-cache.org/about.

[56] WordPress Website. WordPress - accessed 4/2014. https://wordpress.org/.

[57] Windows Azure Service Definition Schema (.csdef) - accessed 3/2014. http://msdn.microsoft.com/en-us/library/windowsazure/ee758711.aspx.

[58] wrk Github Repository - accessed 4/2014. https://github.com/wg/wrk.

[59] L. Zomlot, S. Chandran Sundaramurthy, K. Luo, X. Ou, and S.R. Rajagopalan. Prioritizing intrusion analysis using Dempster-Shafer theory. *In Proceedings of the 4th ACM workshop on Security and artificial intelligence (AISec)*, pages 59–70, 2011.

# An Administrator's Guide to Internet Password Research*

Dinei Florêncio and Cormac Herley
*Microsoft Research, Redmond, USA*

Paul C. van Oorschot
*Carleton University, Ottawa, Canada*

**Abstract.** The research literature on passwords is rich but little of it directly aids those charged with securing web-facing services or setting policies. With a view to improving this situation we examine questions of implementation choices, policy and administration using a combination of literature survey and first-principles reasoning to identify what works, what does not work, and what remains unknown. Some of our results are surprising. We find that offline attacks, the justification for great demands of user effort, occur in much more limited circumstances than is generally believed (and in only a minority of recently-reported breaches). We find that an enormous gap exists between the effort needed to withstand online and offline attacks, with probable safety occurring when a password can survive $10^6$ and $10^{14}$ guesses respectively. In this gap, eight orders of magnitude wide, there is little return on user effort: exceeding the online threshold but falling short of the offline one represents wasted effort. We find that guessing resistance above the online threshold is also wasted at sites that store passwords in plaintext or reversibly encrypted: there is no attack scenario where the extra effort protects the account.

## 1 Introduction

Despite the ubiquity of password-protected web sites, research guidance on the subject of running them is slight. Much of the password literature has become specialized, fragmented, or theoretical, and in places confusing and contradictory. Those who administer and set policies can hardly be blamed for being unenthusiastic about publications which document constantly improving attacks on password sites but are largely silent on the question of how they can be defended. Disappointingly little of the accumulating volume of password research directly addresses key everyday issues—what to do to protect web-services, given the realities of finite-resources, imperfect understanding of the threats, and considerable pushback from users.

Do password composition policies work? Does forced password expiration improve security? Do lockouts help protect a service? What do password meters accomplish? The most comprehensive document on these and other questions dates to 1985 [13]. The problem is not that no recent guidance is available; OWASP offers several documents [39, 45, 56]; blogs, trade magazines and industry analyst reports are full of tips, best practices and opinions. Discussions in online fora reliably generate passionate arguments, if little progress. However, much of the available guidance lacks supporting evidence.

We seek to establish what is supported by clear evidence and solid justification. Using a combination of literature survey and ground-up, first-principles reasoning, we identify what is known to work, what is known not to work, and what remains unknown. The end goal is a more useful view of what is known about the implementation, effectiveness, and impacts of choices made in deploying password-related mechanisms for access to services over the web. The target audience is those interested in the intersection of research literature, and the operation, administration and setting of policies for password-protected web-sites.

On the positive side, considerable progress in the last few years has followed from analysis of leaked plaintext datasets. This has provided new evidence challenging many long-held beliefs. Most current password practices reflect historical origins [18]. Some have evolved over time; others should have, but have not. Environments of use, platforms, and user bases have changed immensely. We summarize the literature useful to answer practical questions on the efficacy of policies governing password composition, expiration and account locking.

Some of our findings are surprising. Experts now recognize that traditional measures of strength bear little relation to how passwords withstand guessing, and

*Proceedings of USENIX LISA'14, Nov. 9-14, 2014, Seattle, WA.

can no longer be considered useful; current password policies have not reflected this. We characterize circumstances allowing advanced offline guessing attacks to occur, and find them more limited than is generally realized. We identify an enormous gap between the guessing-resistance needed to withstand online and offline attacks, and note that it is growing. We highlight that strength above that needed to withstand online guessing is effectively wasted at sites that store passwords in plaintext or reversibly encrypted: there is no attack scenario where the extra strength protects the account from an intelligent adversary.

To dispense with a preliminary question: despite long-known shortcomings in both security and usability, passwords are highly unlikely to disappear. The many reasons include the difficulty of finding something better, user familiarity as an authentication front-end (passwords will likely persist as one factor within multi-dimensional frameworks), and the inertia of ubiquitous deployment [9, 30]. Thus the challenges of administering passwords will not fade quietly either, to the disappointment of those hoping that a replacement technology will remove the need to address hard issues.

## 2 Classifying accounts into categories

A common tactic to allegedly improve the security of password-protected sites is to ask users to expend more effort—choose "stronger" passwords, don't re-use passwords across sites, deploy and administer anti-malware tools, ensure all software on user devices is patched up-to-date, and so on.

If we assume that users have a fixed time-effort budget for "password security" [4], then it is unwise to spend equally on all accounts: some are far more important than others, correspondingly implying greater impact upon account compromise. This motivates determining how to categorize accounts—a subject of surprisingly little focus in the literature. Different categories call for different levels of (password and other) security. Those who decide and administer policies should be aware of what category not only they see their site falling into, but what categories subsets of their users would see it falling into. Note also that some password-protected sites provide no direct security benefit to end-users, e.g., services used to collect data, or which compel email-address usernames to later contact users for marketing-related purposes. Thus views on account and password importance may differ between users and systems administrators or site operators (e.g., see [10]).[1]

**Criteria for categorizing accounts:** A first attempt to categorize password-based accounts might be based on communication technology—e.g., grouping email passwords as one category. We find this unsuitable to our goals, as some email accounts are throw-aways from a "consequences" point of view, while others are critically important. Accounts may be categorized in many ways, based on different criteria. Our categorization (below) is based largely on potential *consequences* of account compromise, which we expect to be important characteristics in any such categorization:

- (personal) time loss/inconvenience
- (personal) privacy
- (personal) physical security
- (personal/business) financial implications
- (personal/business) reputational damage
- (personal/business) legal implications
- confidentiality of third-party data
- damage to resources (physical or electronic)

The above time loss/inconvenience may result from loss of invested effort or accumulated information, such as contact lists in email or social networking accounts. One lens to view this through is to ask: Would a user invest 10 minutes in trying to recover a lost account, or simply create a new such account from scratch? Another account-differentiating question is: Do users make any effort at all to remember the account password? Note also that the consequences of compromise of an account $X$ may extend to accounts $Y$ and $Z$ (e.g., due to password re-use, email-based password recovery for other accounts, accounts used as single-sign-on gateways).

For use in what follows, and of independent interest, we categorize accounts as follows:

- **don't-care** accounts (*unlocked doors*).
- **low-consequence** accounts (*latched garden doors*).
- **medium-consequence** accounts.
- **high-consequence** accounts (*essential/critical*).
- **ultra-sensitive** accounts (*beyond passwords*).

Details and examples of these categories are given in Table 1; we say little more in this paper about the book-end categories in this spectrum: *don't-care* accounts are suitably named, and *ultra-sensitive* accounts are beyond scope. Within this paper, that leaves us to explore what password policies, user advice, implementation details, and security levels are suitable for accounts in three main

[1]Realistic systems administrators might self-categorize their site, asking: Do users see me as a "bugmenot.com" site? (cf. Table 1)

| Category of account | Description | Comments |
|---|---|---|
| 0: Don't-care | Accounts whose compromise has no impact on users. A compromise of the account at any time would not bother users. Often one-use accounts with trivially weak passwords, or recreated from scratch if needed subsequently. Perhaps the site compels passwords, despite user not seeing any value therein. | The security community and users should recognize that for such accounts, there would be no technical objection to using password `password`, knowing that it provides no security. Such accounts should be isolated from other categories to avoid cross-contamination, e.g., due to password re-use. Users should minimize security-related investments of time and effort—resources are better spent elsewhere. Possible strategies: re-using a single weak password for all such accounts, using distinct passwords written down on one sheet for easy access, and using publicly shared passwords (see: bugmenot.com). |
| | **Generic examples**: One-time email accounts (e.g., used for one-off signup, then abandoned). Nuisance accounts for access to "free" news articles or other content. | |
| 1: Low-consequence | Accounts whose compromise has non-severe implications (minimal or easily repaired). Often infrequently used accounts, relatively low-impact if compromised. | Administrators and operators should be realistic in expectations of user commitment to such accounts. Some users may rely almost entirely on a password recovery feature, vs. remembering such account passwords. Users should recognize the place of these between *Don't-care* and *Medium-consequence* accounts. |
| | **Generic examples**: Social network accounts (infrequent users). Discussion group accounts (infrequent users). Online newspapers, streaming media accounts (credit card details not stored onsite). | |
| 2: Medium-consequence | Non-trivial consequences but limited, e.g., loss of little-used reputation account, or credit card details stored at online U.S. merchant (direct financial losses limited to $50). | User losses are more in time and effort, than large financial loss or confidentiality breached by document or data loss. User effort directed towards resisting online guessing attacks is well-spent. Unclear if the same holds true re: resisting determined offline guessing attacks. Note: many attack vectors are beyond user control (e.g., browser-based flaws, server compromises). |
| | **Generic examples**: Email accounts (secondary). Online shopping sites (credit card details stored onsite). Social network accounts (casual users). Voice or text communication services accounts (e.g., Skype, MSN). Charge-up sites for stored value cards (credit card details stored onsite). Human resources sites giving employees semi-public information. | |
| 3: High-consequence | Critical or essential accounts related to primary employment, finance, or documents requiring high confidentiality. Compromises are not easily repaired or have major consequences/side effects. | Most important password discussion, attention and effort of both sysadmins and users should focus here. Often password protection for such accounts is best augmented by second-factor mechanisms (involving explicit user action) or other dimensions (invisible to user). Stakeholder priorities may differ: an account viewed lower consequence by a user may be categorized essential by their employer (e.g., remote access to a corporate database via a password). |
| | **Generic examples**: Email accounts (primary, professional, recovery of other accounts). Major social network/reputational accounts (heavy users and celebrities). Online banking and financial accounts. SSH and VPN passwords for access to corporate networks. Access to corporate databases, including employee personal details. | |
| $\infty$: Ultra-sensitive | Account compromise may cause major, life-altering, irreversible damage. (Many individual users will have no such accounts.) | It is entirely unsuitable to rely on passwords alone for securing such accounts Passwords if used should be augmented by (possibly multiple) additional mechanisms. The passwords themselves should not be expected to be tangibly different from those for high-consequence accounts (one might argue that weaker passwords suffice, given stronger supplementary authentication mechanisms). |
| | **Generic examples**: Multi-million dollar irreversible banking transactions. Authorization to launch military weapons. Encryption of nation-state secrets. | |

Table 1: Categories of password-protected accounts, comments and examples. Accounts in the same category ideally have passwords of similar strength relative to guessing attacks. Ultra-sensitive accounts require that passwords be augmented by much more robust mechanisms.

categories of interest: *low-consequence, medium consequence, and high-consequence* accounts.[2]

**Use of single term "password" over-loaded?** Our discussion of categories highlights that using the unqualified term *password* for protection that runs the gamut from don't-care to high-consequence sites may mislead users. We should not be quick to express outrage on learning that `password1` and `123456` are common on publicly-disclosed password lists from compromised sites, if these are don't-care accounts in users' eyes. Nor should it be surprising to find passwords stored cleartext on fantasy football sites. The use of the same term *password* across all account categories, together with a jumble of unscoped password advice to users, and an absence of discussion of different categories of accounts (and corresponding password requirements), likely contributes to lower overall security, including through cross-category re-use of passwords. We believe finer-grained terminology would better serve users here.

## 3 Guessing attacks and password storage

The enormous effort that has been spent on password strength and guessing-attacks might lead us to believe that the questions there are largely settled and things are well-understood. Unfortunately, we find that this is not the case. For a number of reasons, realistic studies of password behaviors are hard to conduct [21].

Until recently, published knowledge on in-the-wild password habits [7, 10, 57] was derived from a few small-scale sets of plaintext passwords [38], or studies without access to plaintext passwords [22]. Recent large-scale breaches have provided significant collections of plaintext passwords, allowing study of actual user choices. Tellingly, they reveal that many time-honored assumptions are false. Password "strength" measures both long-used by academics and deeply embedded in criteria used by IT security auditors, are now known to correlate poorly with guessing resistance; policies currently enforced push users toward predictable strategies rather than randomness—e.g., evidence, as discussed below, shows password aging (forced expiration) achieves very little of its hoped-for improvement [63].

Here we explore what is known on measuring password strength, fundamentals of storing passwords, and suitable target thresholds for how many guesses a password should withstand.

**Leaked datasets.** Table 2 lists several recent leaks from prominent web-sites. These datasets reveal much about user password habits. The ethics of doing analysis on what is, in effect, stolen property generated some discussion when the Rockyou dataset became available in 2009. While none of the 32 million users gave permission for their passwords to be used, rough consensus now appears to be that use of these datasets imposes little or no additional harm, and their use to improve password security is acceptable; the datasets are also of course available to attackers.

Note that among Table 2 incidents, passwords were stored "properly" (see Section 3.4) salted and hashed in just two cases—Evernote and Gawker. Rockyou, Tianya and Cupid Media stored plaintext passwords; LinkedIn and eHarmony stored them hashed but unsalted; Adobe stored them reversibly encrypted. Section 3.2 and Figure 1 explain why offline guessing attacks (beyond rainbow table lookups) are a relevant threat only when the password file is properly salted and hashed.

### 3.1 Password strength: ideal vs. actual

Security analysis and evaluation would be much simpler if users chose passwords as random collections of characters. For example, if passwords were constrained to be $L$ characters long, and drawn from an alphabet of $C$ characters, then each of $C^L$ passwords would be equally likely. An attacker would have no better strategy than to guess at random and would have probability $C^{-L}$ of being correct on each guess. Even with relatively modest choices for $L$ and $C$ we can reduce the probability of success (per password guess) to $10^{-16}$ or so—putting it beyond reach of a year's worth of effort at 10 million guess verifications per second.

Unfortunately, the reality is nowhere close to this. Datasets such as the 32 million plaintext Rockyou passwords [64] have revealed that user behavior still forms obvious clusters three decades after attention was first drawn to the problem [38]. Left to themselves users choose common words (e.g., `password, monkey, princess`), proper nouns (e.g., `julie, snoopy`), and predictable sequences (e.g., `abcdefg, asdfgh, 123456`; the latter by about 1% of Rockyou accounts).

This has greatly complicated the task of estimating passwords' resistance to guessing. Simple estimation techniques work well for the ideal case of random collections of characters, but are completely unsuited for user-chosen passwords. For example, a misguided approach models the "entropy" of the password as $\log_2 C^L = L \cdot \log_2 C$ where $L$ is the length, and $C$ is the size of the alphabet from which the characters are drawn (e.g., lowercase only would have $C = 26$, lowercase and digits would have $C = 36$, lower-, uppercase and digits would have $C = 62$ etc). The problems with this approach becomes obvious when we factor in user behavior: `P@ssw0rd` occurs 218 times in the Rock-

[2]An alternate account categorization by Grosse and Upadhyay [28], based on *value*, has categories: throw-away, routine, spokesperson, sensitive and very-high-value transactions.

| Site | Year | # Accounts | Hashed | Salted | Reversibly Encrypted | Offline guessing attack beyond rainbow tables needed and possible |
|---|---|---|---|---|---|---|
| Rockyou [64] | 2009 | 32m | | | | N |
| Gawker | 2010 | 1.3m | ✓ | ✓ | | Y |
| Tianya | 2011 | 35m | | | | N |
| eHarmony | 2012 | 1.5m | ✓ | | | N |
| LinkedIn | 2012 | 6.5m | ✓ | | | N |
| Evernote | 2013 | 50m | ✓ | ✓ | | Y |
| Adobe | 2013 | 150m | | | ✓ | N |
| Cupid Media | 2013 | 42m | | | | N |

Table 2: Recent incidents of leaked password data files. For only two listed incidents (Evernote and Gawker) would an offline guessing attack be the simplest plausible way to exploit the leak. For each other incident, passwords were stored in such a way that either an easier attack would suffice, or offline guessing was impossible, as explained in Figure 1.

you dataset but has a score of 52.6 under this measure, while `gunpyo` occurs only once and has score 28.2. Thus, a password that is far more common (thus more likely to be guessed) scores much higher than one that is unique—the opposite of what a useful metric would deliver. These are by no means exceptions; a test of how passwords hold up to guessing attacks using the JohntheRipper cracking tool [44] shows that $L \cdot \log_2 C$ correlates very poorly with guessing resistance [32, 60]. Similarly for NIST's crude entropy approximation [13, 60]: many "high-entropy" passwords by its measure turn out to be easily guessed, and many scoring lower withstand attack quite well. Such naive measures dangerously and unreliably estimate how well passwords can resist attack.

The failure of traditional measures to predict guessability has led researchers to alternatives aiming to more closely reflect how well passwords withstand attack. One approach uses a cracking tool to estimate the number of guesses a password will survive. Tools widely used for this purpose include JohntheRipper [44], Hashcat, and its GPU-accelerated sister oclHashcat [42]; others are based on context-free grammars [60, 61]. These tools combine "dictionaries" with *mangling rules* intended to mimic common user strategies: replacing '`s`' with '`$`', '`a`' with '`@`', assume a capital first letter and trailing digit where policy forces uppercase and digits, etc. Thus simple choices like `P@ssw0rd` are guessed very quickly.

Another approach models an optimal attacker with access to the actual password distribution $\chi$ (e.g., the Rockyou dataset), making guesses in likelihood order. This motivates *partial guessing metrics* [6] addressing the question: how much work must an optimal attacker do to break a fraction $\alpha$ of user accounts?

Bonneau's $\alpha$-guesswork gives the expected number of guesses per-account to achieve a success rate of $\alpha$ [7]. Optimal guessing gives dramatic improvements in skewed distributions arising from user-chosen secrets, but none in uniform distributions. For example, for the distribution $U_{L6}$ of random (equi-probable) length-6 lowercase passwords, all can be found in $26^6 \approx 309$ million guesses per account, and 10% in 30.9 million guesses. In contrast, guessing in optimal order on the Rockyou distribution of 32 million passwords, an average of only $7,131$ guesses per account breaks 10% of accounts. Thus, successfully guessing 10% of the Rockyou accounts is a factor of $30,900,000/7131 \approx 4300$ easier than for a length-6 lowercase random distribution (even though the latter is weaker using the $L \cdot \log_2 C$ measure).

Thus, oversimplified "entropy-based" measures should not be relied upon to draw conclusions about guessing resistance; rather, their use should be strongly discouraged. The terms password *strength* and *complexity* are also confusing, encouraging optimization of such inappropriate metrics, or inclusion of certain character classes whether or not they help a password withstand attack. We will use instead the term *guessing resistance* for the attribute that we seek in a password.

## 3.2 Online and offline guessing

Determining how well a password withstands attack requires some bound on how many guesses the attacker can make and some estimate of the order in which he makes them. The most conservative assumption on guessing order is that the attacker knows the actual password distribution (see $\alpha$-guesswork above); another approach assumes that he proceeds in the order dictated by an efficient cracker (see also above). We now review how the number of guesses an attacker can make depends on both: (a) the point at which he attacks; and (b) server-side details of how passwords are stored.

The main points to attack a password are: on the client, in the network, at a web-server's public-facing part, and at the server backend. Attacks at the client (e.g., malware, phishing) or in the network (e.g., sniffing) do not generally involve guessing—the password is simply stolen; guess-resistance is irrelevant. Attacks involving guessing are thus at the server's public-face and backend.

Attacks on the server's public-face (generally called *online attacks*) are hard to avoid for a public site—by design the server responds to authentication requests, checking (username, password) pairs, granting access when they match. An attacker guesses credential pairs and lets the server to do the checking. Anyone with a browser can mount basic online guessing attacks on the publicly facing server—but of course the process is usually automated using scripts and guessing dictionaries.

Attacks on the backend are harder. Recommended practice has backends store not passwords but their salted hashes; recalculating these from user-entered passwords, the backend avoids storing plaintext passwords. (Sections 3.4 and 3.5 discuss password storage details.)

For an offline attack to improve an attacker's lot over guessing online, three conditions must hold.

i) He must gain access to the system (or a backup) to get to the stored password file. Since the backend is designed to respond only one-at-a-time to requests from the public-facing server, this requires evading all backend defences. An attacker able to do this, and export the file of salted-hashes, can test guesses at the rate his hardware supports.

ii) He must go undetected in gaining password file access: if breach detection is timely, then in well-designed systems the administrator should be able to force system-wide password resets, greatly limiting attacker time to guess against the file. (Note: ability to quickly reset passwords requires nontrivial planning and resources which are beyond scope to discuss.)

iii) The file must be properly both salted and hashed. Otherwise, an offline attack is either not the best attack, or is not possible (as we explain next).

If the password file is accessed, and the access goes undetected, then four main possibilities exist in common practice (see Figure 1):

1) *the file is plaintext*. In this case, offline guessing is clearly unnecessary: the attacker simply reads all passwords, with nothing to guess [64].

2) *the file is hashed but unsalted*. Here the passwords cannot be directly read, but *rainbow tables* (see below) allow fast hash reversal for passwords within a fixed set for which a one-time pre-computation was done. For example, 90% of LinkedIn's (hashed but unsalted) passwords were guessed in six days [26].

3) *the file is both salted and hashed*. Here an offline attack is both possible and necessary. For each guess against a user account the attacker must compute the salted hash and compare to the stored value. The fate of each password now depends on how many guesses it will withstand.

| Position | Password | Per-guess success probability |
|---|---|---|
| $10^0$ | `123456` | $9.1 \times 10^{-3}$ |
| $10^1$ | `abc123` | $5.2 \times 10^{-4}$ |
| $10^2$ | `princesa` | $1.9 \times 10^{-4}$ |
| $10^3$ | `cassandra` | $4.2 \times 10^{-5}$ |
| $10^4$ | `sandara` | $5.3 \times 10^{-6}$ |
| $10^5$ | `yahoo.co` | $7.2 \times 10^{-7}$ |
| $10^6$ | `musica17` | $9.4 \times 10^{-8}$ |
| $10^7$ | `tilynn06` | $3.1 \times 10^{-8}$ |

Table 3: Passwords from the Rockyou dataset in position $10^m$ for $m = 0, 1, 2, \cdots, 7$. Observe that an online attacker who guesses in optimal order sees his per-guess success rate fall by five orders of magnitude if he persists to $10^6$ guesses.

4) *the file has been reversibly encrypted*. This case has two paths: the attacker either gets the decryption key (Case 4A), or he does not (Case 4B).

In Case 4A, offline attack is again unneeded: decryption provides all passwords. In Case 4B, there is no effective offline attack: even if the password is `123456`, the attacker has no way of verifying this from the encrypted file without the key (we assume that encryption uses a suitable algorithm, randomly chosen key of sufficient length, and standard cryptographic techniques, e.g., initialization, modes of operation, or random padding, to ensure different instances of the same plaintext password do not produce identical ciphertext [36]). Even having stolen the password file and exported it without detection, the attacker's best option remains online guessing at the public-facing server; properly encrypted data should not be reversible, even for underlying plaintext (passwords here) that is far from random.

In summary, Figure 1 shows that offline guessing is a primary concern only in the narrow circumstance when all of the following apply: a leak occurs, goes undetected,[3] and the passwords are suitably hashed and salted (cf. [8, 64]). In all other common cases, offline attack is either impossible (guessing at the public-facing server is better) or unneeded (the attacker gets passwords directly, with no guessing needed).

Revisiting Table 2 in light of this breakdown, note that of the breaches listed, Evernote and Gawker were the only examples where an offline guessing attack was necessary; in all other cases a simpler attack sufficed, and thus guessing resistance (above that necessary to resist online attack) was largely irrelevant due to how passwords were stored.

**Rainbow tables.** To understand the importance of salting as well as hashing stored passwords, consider the attacker wishing to reverse a given hashed password. Starting with a list (dictionary) of $N$ candidate passwords, pre-computing the hash of each, and stor-

[3]Or similarly, password reset capability is absent or unexercised.

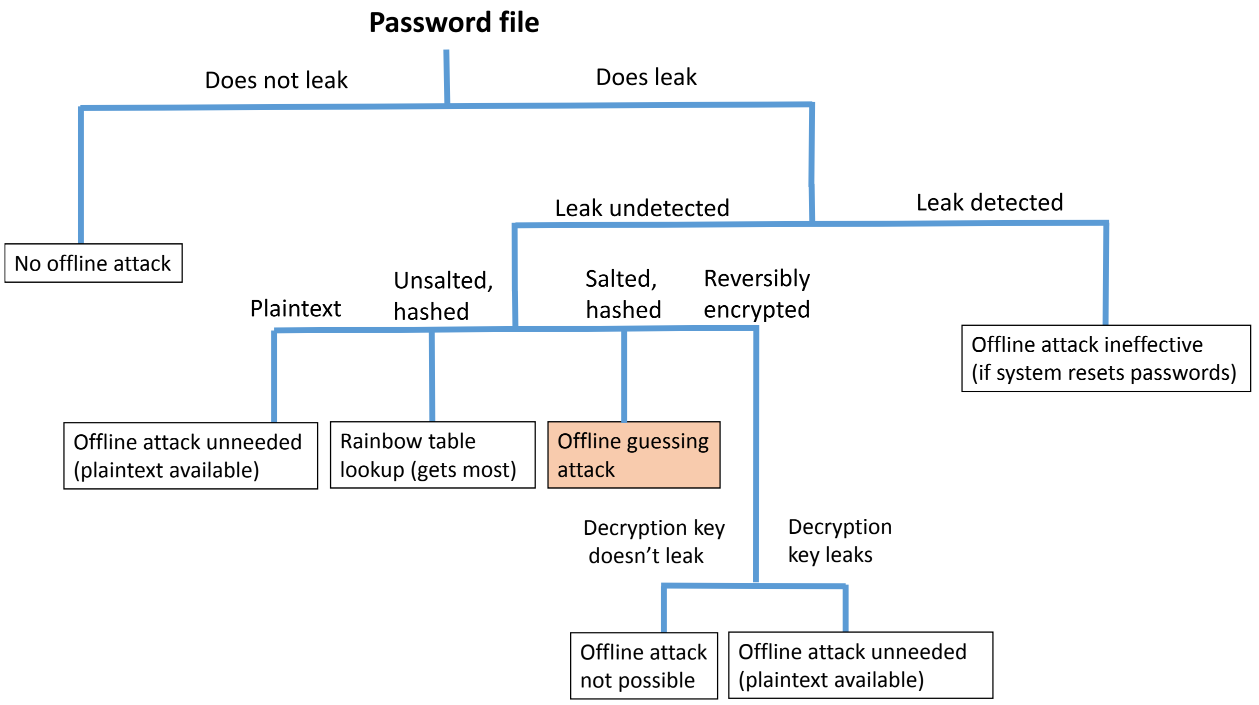

Password file
Does not leak
Does leak
No offline attack
Leak undetected
Leak detected
Plaintext
Unsalted, hashed
Salted, hashed
Reversibly encrypted
Offline attack ineffective (if system resets passwords)
Offline attack unneeded (plaintext available)
Rainbow table lookup (gets most)
Offline guessing attack
Decryption key doesn't leak
Decryption key leaks
Offline attack not possible
Offline attack unneeded (plaintext available)

Figure 1: Decision tree for guessing-related threats in common practice based on password file details. Offline guessing is a threat when the password file leaks, that fact goes undetected, and the passwords have been properly salted and hashed. In other cases, offline guessing is either unnecessary, not possible, or addressable by resetting system passwords. Absent a hardware security module (HSM), we expect that the "Decryption key doesn't leak" branch is rarely populated; failure to prevent theft of a password file gives little confidence in ability to protect the decryption key.

| Length | Character set | Full cardinality |
|---|---|---|
| 12 | lower | $26^{12} = 2^{56.4}$ |
| 10 | lower, upper | $52^{10} = 2^{57.0}$ |
| 9 | any | $95^{9} = 2^{59.1}$ |
| 10 | lower, upper, digit | $62^{10} = 2^{59.5}$ |

Table 4: Number of elements targeted by various rainbow tables.

ing each pair sorted by hash, allows per-instance reversal by simple table lookup after one-time order-$N$ precomputation—and order-$N$ storage. To reduce storage, *rainbow tables* [43] use a series of functions to precompute repeatable sequences of password hashes called *chains*, storing only each chain's first and last plaintext. Per-instance computation later identifies hashes from the original fixed password list to a chain, allowing reversal in greater, but still reasonable, time than had all hashes been stored. Numerous rainbow table implementations and services are available.[4] For rainbow tables targeting selected passwords compositions, Table 4 lists as reference points the targeted number of passwords, which give a lower bound on pre-computation time (resolving expected "collisions" increases computation time).

Modifications [40] may allow tables for any efficiently enumerable password space, e.g., based on regular expressions for defined patterns of lower, upper, digits and special characters; this would extend attacks from (naive) brute-force spaces to "smart dictionaries" of similar size but containing higher-likelihood user passwords. We emphasize that *offline attacks using pre-computations over fixed dictionaries, including rainbow tables, are defeated by proper salting, and require leaked password hashes.*

[4]For example, see `http://project-rainbowcrack.com` or `sourceforge.net/projects/ophcrack` among others.

## 3.3 How many guesses must a password withstand?

Recall that the online attacker's guesses are checked by the backend server, while an offline attacker tests guesses on hardware that he controls. This constrains online attacks to far fewer guesses than is possible offline.

**Online guessing (breadth-first).** Consider the online attacker. For concreteness, assume a guessing campaign over a four-month period, sending a guess every 1s at a sustained rate, yielding about $10^7$ guesses; we use this as a very loose upper bound on the number of online guesses any password might have to withstand. An attacker sending guesses at this rate against all accounts (a breadth-first attack) would likely simply overwhelm servers: e.g., it is unlikely that Facebook's servers could handle simultaneous authentication requests from all users. (In practice, but a tiny fraction authenticate in any 1s period.) Second, if we assume that the average user attempts authentication $k$ times/day and fails $5\%$ of the time (due to typos, cached credentials after a password change, etc.) then a single attacker sending one guess per-account per-second would send $86,000/k$ times more traffic and $1.73 \times 10^7/k$ more fail events than the entire legitimate user population combined. Even if $k = 100$ (e.g., automated clients re-authenticating every 15 minutes) our single attacker would be sending a factor of $860$ more requests and $1.73 \times 10^5$ more fails than the whole legitimate population. Malicious traffic at this volume against any server is hard to hide. A more realistic average of $k = 1$ makes the imbalance between malicious and benign traffic even more extreme. Thus $10^7$ guesses per account seems entirely infeasible in a breadth-first online guessing campaign; $10^4$ is more realistic.

**Online guessing (depth-first)**. What about depth-first guessing—is $10^7$ guesses against a single targeted account feasible? First, note that most individual accounts are not worthy of targeted effort. Using the Section 2 categories, low- and medium-consequence sites may have very few such accounts, while at high-consequence sites a majority might be worthy. Second, $10^7$ guesses would imply neither a lockout policy (see Section 4.4) nor anything limiting the rate at which the server accepts login requests for an account. Third, as evident from Table 3 which tabulates the passwords in position $10^m$ from the Rockyou distribution for $m = 0, 1, 2, \cdots, 7$, an online attacker making guesses in optimal order and persisting to $10^6$ guesses will experience five orders of magnitude reduction from his initial success rate. Finally, IP address blacklisting strategies may make sending $10^7$ guesses to a single account infeasible (albeit public IP addresses fronting large numbers of client devices complicate this). Thus, the assumptions that would allow $10^7$ online guesses against a single account are extreme—effectively requiring an absence of defensive effort. $10^6$ seems a more realistic upper bound on how many online guesses a password must withstand in a depth-first attack (e.g., over 4 months). This view is corroborated by a 2010 study of password policies, which found that Amazon.com, Facebook, and Fidelity Investments (among many others) allow 6-digit PIN's for authentication [23]. That these sites allow passwords which will not (in expectation) survive $10^6$ guesses suggests that passwords that will survive this many guesses can be protected from online attacks (possibly aided by backend defenses). Figure 2 depicts our view: we gauge the online guessing risk to a password that will withstand only $10^2$ guesses as extreme, one that will withstand $10^3$ guesses as moderate, and one that will withstand $10^6$ guesses as negligible. The left curve does not change as hardware improves.

**Offline guessing**. Now consider the offline attacker: using hardware under his control, he can test guesses at a rate far exceeding online attacks. Improvements in processing power over time make it possible that his new hardware computes guesses orders of magnitude faster than, say, 10-year-old authentication servers which process (online) login attempts. The task is also distributable, and can be done using a botnet or stolen cloud computing resources. An attacker might use thousands of machines each computing hashes thousands of times faster than a target site's backend server. Using a GPU able to compute 10 billion raw hashes/s or more [18, 26], a 4-month effort yields $10^{17}$ guesses; 1,000 such machines allows $10^{14}$ guesses on each of a million accounts, or $10^{20}$ on a single account—all assuming no defensive iterated hashing, which Section 3.4 explores as a means to reduce such enormous offline guess numbers.

Given the lack of constraints, it is harder to bound the number of guesses, but it is safe to say that offline attacks can test many orders of magnitude more guesses than online attacks. Weir et al. [60] pursue cracking up to $10^{11}$ guesses; a series of papers from CMU researchers investigate as far as $10^{14}$ guesses [32, 35]. To be safe from offline guessing, we must assume a lower bound of at least $10^{14}$, and more as hardware[5] and cracking methods improve. This is illustrated in Figure 2.

**Online-offline gap**. To summarize, a huge chasm separates online and offline guessing. *Either* an attacker sends guesses to a publicly-facing server (online) *or* guesses on hardware he controls (offline)—there is no continuum of possibilities in between. The number of guesses that a password must withstand to expect to survive each attack differs enormously. A threshold of at most $10^6$ guesses suffices for high probability of surviving online attacks, whereas at least $10^{14}$ seems necessary for any confidence against a determined, well-resourced offline attack (though due to the uncertainty about the attacker's resources, the offline threshold is harder to estimate). These thresholds for probable safety differ by 8 orders of magnitude. The gap increases if the offline attack brings more distributed machines to bear, and as offline attacks and hardware improve; it decreases with hash iteration. Figure 2 conceptualizes the situation and Table 5 summarizes.

Next, consider the incremental benefit received in improving a password as a function of the number of guesses it can withstand ($10^m$). Improvement delivers enormous gain when $m \leq 3$: the risk of online attack is falling sharply in this region, and safety (from online guessing) can be reached at about $m = 6$. By the time $m = 6$, this effect is gone; the risk of online attack is now minimal, but further password improvement buys little protection against offline attacks until $m = 14$ (where

[5]Hardware advances can be partially counteracted by increased hash iteration counts per Section 3.4.

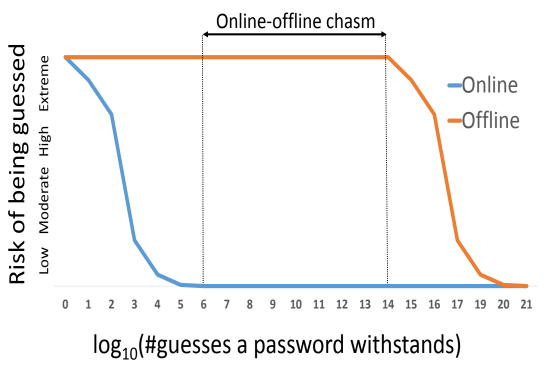


Figure 2: Conceptualized risk from online and offline guessing as a function of the number of guesses a password will withstand over a 4-month campaign. In the region from $10^6$ to about $10^{14}$, improved guessing-resistance has little effect on outcome (online or offline).

the probability of offline guessing success starts to decline). Note the large gap or *chasm* where online guessing is a negligible threat but surviving offline guessing is still far off. *In this gap, incrementally increasing the number of guesses the password will survive delivers little or no security benefit.*

For example, consider two passwords which withstand $10^6$ and $10^{12}$ guesses respectively. Unless we assume the offline attacker lacks motivation or resources (and gives up early), there is no apparent scenario in which the extra guess-resistance of the second password helps. For example, a password like `tincan24` (which will survive more than a million guesses derived from the Rockyou distribution) and one like `7Qr&2M` (which lives in a space that can be exhausted in $(26 + 26 + 10 + 30)^6 = 92^6 < 10^{12}$ guesses) fare the same: both will survive online guessing, but neither will survive offline attack. Equally, a 7-digit and a 13-digit random PIN have similar security properties for the same reason. If we assume additional guess-resistance comes at the cost of user effort [4, 25, 29], then the effort in the second case appears entirely wasted. In this case, *an effort-conserving approach is to aim to withstand online attacks, but not put in the extra effort to withstand offline attacks.* In fact there is evidence that many sites abandon the idea of relying on user effort as the defence against offline attacks; i.e., they appear to make little effort to force users to reach the higher threshold [23]. Sections 4.1 and 4.2 consider the efficacy of password composition policies and blacklists.

**Frequency of online vs. offline attacks**. Authoritative statistics on the relative frequency of online attacks compared to offline attacks do not exist. However it is clear that online attacks can be mounted immediately against public web sites (such attacks are more efficient when known-valid userids are obtained a priori [24]), while offline attacks require that the password hashes be available to the attacker (e.g., leaked file).

## 3.4 Storage and stretching of passwords

As Section 3.2 stated, storing salted hashes [48] of passwords beats most alternatives. In all other common storage options, advanced offline guessing attacks are either unnecessary (simpler attacks prevail) or impossible. Use of site-wide (global) salt defeats generic rainbow tables; per-account salts (even userid) and iteration counts, storable with hashes, provide further protection. Unfortunately, for various reasons, hashing is far from universal (e.g., perhaps 40% of sites do not hash [10]).

Guessing is resource-intensive—an offline attack may involve billions of guesses per-account, whereas a webserver verifies login attempts only on-demand as users seek to log in. Since early UNIX, this has been leveraged defensively by hash functions designed to be slow, by iteration: repeatedly computing the hash of the hash of the salted password. Such *key stretching* was formally studied by Kelsey et al. [33]; we reserve the term *key strengthening* for the idea, with related effect, of using a random suffix salt that verifiers must brute-force. Iterating $10^n$ times slows offline attack by $n$ orders of magnitude; this configurable factor should be engineered to add negligible delay to users, while greatly increasing an offline attacker's work. Factors of 4,000–16,000 iterations already appear in practice. Our estimates for the number of guesses an offline attacker can send assumed no iteration; hash iteration narrows the online-offline chasm.

Salting also removes one form of "parallel attack": if two users have the same password, this will not be apparent and cannot be exploited to simplify attacks (assuming proper salting and hashing, e.g., salts of sufficient length, and passwords not truncated before hashing).

Practical instantiations [18] of key stretching (via so-called *adaptive key derivation functions*) include `bcrypt` [48], supported first in OpenBSD with 128-bit salt and configurable iteration count to acceptably adjust delay and server load on given platforms, and allow for hardware processing advances; the widely used standard `PBKDF2` (part of PKCS #5 v2.0 and RFC 2898); and the newer `scrypt` [46] designed to protect against custom-hardware attacks (e.g., ASIC, FPGA, GPU).

**Keyed hashing.** Reversible encryption is one of the worst options for storing passwords if the decryption key leaks, but is among the best if a site can guarantee that it never leaks (even if the password file itself does). Justification for sites to store passwords reversibly encrypted is a need to support legacy protocols (see Section 3.5). Absent such legacy requirements, the best solution is salting and iterated hashing with a message authentication code (MAC) [37, 56] stored instead of a hash; password verification (and testing of guesses) is then impossible without crypto key access. The difficulty of managing keys should not be understated—too often keys stored in software or a configuration file are found by attackers, explaining the common use of a one-way hash over reversible encryption. However, if the MAC of a salted, iterated password hash is all that is stored, then even if the MAC key leaks, security is equal to a salted iterated hash; and that risk falls away if a hardware security module (HSM) is used for MAC generation and verification.

## 3.5 Availability of passwords at the server

So salted hashes are a preferred means to store passwords, and (cf. Figure 1) an attacker who has access to the password file, and exports it undetected, still faces a computationally expensive offline attack. A site suffering this severe, undetected breach fares far better than one with plaintext or hashed-unsalted passwords, or reversibly encrypted passwords and a leaked decryption key. Nonetheless, many sites use a non-preferred means of storing passwords, e.g., there is a "Hall of Shame" of sites[6] which mail forgotten passwords back to users and thus store them either plaintext or reversibly encrypted. While the practice is inadvisable for high-consequence sites, as Section 2 notes, one size clearly does not fit all.

In addition to sites which mail-back passwords, recent breaches clearly signal that storing plaintext passwords is not uncommon. In Table 2's list of recent server leaks, only two used salted hashes. Failure to store passwords as salted hashes may be due to confusion, failure to understand the advantages, or a conscious decision or software default related to legacy applications or protocols as we explain next.

RADIUS (Remote Authentication Dial In User Service) is a networking protocol widely used to provide dial-in access to corporate and university networks. Early protocols that allowed client machines to authenticate such as Password Authentication Protocol (PAP) and Challenge-Handshake Authentication Protocol (CHAP) over RADIUS require passwords be available (to the server) in-the-clear or reversibly encrypted. Thus, sites that supported such clients must store passwords plaintext or reversibly encrypted. Support for protocols that supercede PAP and CHAP in commodity OS's began only circa 2000. Thus, many sites may have had to support such clients at least until a decade or so later.

Universities provide interesting examples. Recent papers by groups researching passwords make clear that several universities were, at least until recently, storing passwords reversibly encrypted or as unsalted hashes. Mazurek et al. (CMU) state [35]: "The university was using a legacy credential management system (since abandoned), which, to meet certain functional requirements, reversibly encrypted user passwords, rather than using salted, hashed records." Fahl et al. (Leibniz University) state [21]: "The IDM system stored up to five unique passwords per user using asymmetric cryptography, so it would be possible to decrypt the passwords to do a security analysis." Zhang et al. (UNC) state [63]: "The dataset we acquired contains 51,141 unsalted MD5 password hashes from 10,374 defunct ONYENs (used between 2004 and 2009), with 4 to 15 password hashes per ONYEN, i.e., the hashes of the passwords chosen for that ONYEN sequentially in time."[7]

Figure 1 makes clear that if the password is to be available at the backend (i.e., stored plaintext or reversibly encrypted) then an offline attack is either unnecessary or impossible. Thus, any resistance to guessing above and beyond that needed to withstand online attacks is wasted (in no scenario does the extra guessing resistance protect the account from competent attackers). Thus sites that impose restrictive password policies on their users while storing passwords plaintext or reversibly encrypted are squandering effort. An example appears to be a documented CMU policy [35]: passwords had to be greater than length 8 and include lower, upper, special characters and digits. This policy appears designed to withstand an offline guessing attack which (since passwords were reversibly encrypted) had no possibility of occurring, and thus imposes usability cost without security benefit.

We do not know how common it is for sites to store passwords plaintext or reversibly encrypted. Large breaches, such as in Table 2, continue to make clear that plaintext is common among low- and medium-consequence sites. The data from CMU and Leibniz hint that far from being rare exceptions, reversible encryption of passwords may also be quite common. If true, this would imply that many sites with strict composition policies are engaged in a large-scale waste of user effort based on confused thinking about guessing resistance.

## 3.6 Other means to address offline attacks

Online guessing attacks seem an unavoidable reality for Internet assets protected by passwords, while offline attacks occur only in a limited set of circumstances. The guessing resistance needed to withstand these two types of attacks differs enormously (recall Section 3.3). Significant effort has been devoted to getting users to choose better passwords. If an online attacker can send at most $10^6$ guesses per account, then it is relatively easy (e.g., password blacklists) to resist online guessing. Thus, getting users to choose passwords that will withstand over $10^6$ guesses is an effort to withstand offline attacks, not online.

[6] See `http://plaintextoffenders.com`.

[7] An "ONYEN" is a userid ("Only Name You'll Ever Need") in the single-sign-on system studied.

There are ways to address offline attacks that do not involve persuading users to choose better passwords. Figure 1 makes clear that if the file doesn't leak, or the leak is detected and existing passwords are immediately disabled, things are very different. Thus alternate approaches include those that protect the password file, or allow detection of leaks—neither requiring changes in user behaviour.

Crescenzco et al. [15] give a method to preclude an offline attack, even if an attacker gains unrestricted access to the backend server. It hinges on the fact that an offline attacker must guess at a rate far exceeding the normal authentication requests from the user population (cf. Section 3.3). They introduce a novel hashing algorithm that requires randomly indexing into a large collection of random bits (e.g., 1 TByte). Ensuring that the only physical connection to the server with the random bits is matched to the expected rate of authentication requests from the user population guarantees that the information needed to compute the hashes can never be stolen. While the scheme is not standard, it illustrates that ingenious approaches to prevent password file leaks are possible (thereby eliminating the possibility of offline attacks).

Leaked password files can also be detected by spiking password files with *honeywords*—false passwords which are salted, hashed and indistinguishable from actual user passwords [31]. An offline attack which attempts authentication with a "successfully" guessed honeyword alerts administrators of a breached password file, signalling that in-place recovery plans should commence.

## 4 Password policies and system defences

### 4.1 Composition and length policies

Many approaches have been tried to force users to choose better passwords. The most common are policies with length and character-group composition requirements. Many sites require passwords of length at least 8, with at least three of four character types (lower- and uppercase, digits, special characters) so that each password meets a lower bound by the measure $L \cdot \log_2 C$. However, as Section 3.1 explains, this naive entropy-motivated metric very poorly models password guessing-resistance [32, 60]. Users respond to composition policies with minimally compliant choices such as `Pa$$w0rd` and `Snoopy2`. Passwords scoring better by this metric are not guaranteed to fare better under guessing attacks. On examining this, Weir et al. [60] conclude "the entropy value doesn't tell the defender any useful information about how secure their password creation policy is."

Recent gains in understanding guess-resistance come largely from analysis of leaked datasets [7, 60]. Since it appears (including by examining the actual cleartext passwords) that none of Table 2's listed sites imposed strict composition policies on users, we cannot directly compare collections of passwords created with and without composition polices to see if the policy has a significant effect. However, Weir et al. [60] compare how subsets of the Rockyou dataset that comply with different composition policies fare on guessing resistance. (The exercise is instructive, but we must beware that a subset of passwords that comply with a policy are not necessarily representative of passwords created under that policy; cf. [32].) They found that passwords containing an uppercase character are little better at withstanding guessing than unrestricted passwords: 89% of the alpha strings containing uppercase were either all uppercase, or simply had the first character capitalized (cf. [35]). They conclude that forcing an uppercase character merely doubles the number of guesses an intelligent attacker would need. Fully, 14% of passwords with uppercase characters did not survive 50,000 guesses—thus providing inadequate protection even against online attackers.

Including special characters helped more: of passwords incorporating one, the number that did not survive 50,000 guesses dropped to 7%. But common patterns revealed by their analysis (e.g., 28.5% had a single special character at the end) were not fully exploited by the guessing algorithm, so this survival rate is optimistic. Thus including special characters likewise does not protect robustly even against online attacks.

Kelley et al. [32] examine passwords created by 12,000 participants in a Mechanical Turk study under 8 different composition policies including: basic-length-8, basic-length-16, and length-8 mandating all of lower, upper, digits and special characters. They use a variety of cracking algorithms to evaluate guessing resistance of various passwords. Interestingly, while there is enormous variation between the fate of passwords created under different policies at high guess numbers (e.g., 58% of basic-length-8, but only 13% of basic-length-16 passwords were found after $10^{13}$ guesses) there was less variation for numbers of guesses below $10^6$. Also, in each of the policies tested, fewer than 10% of passwords fell within the first $10^6$ guesses (our online threshold).

Mazurek et al. [35] examine 25,000 passwords (from a university single sign-on system) created under a policy requiring at least length-8 and mandating inclusion of lower, upper, special characters and digits, and checks against a dictionary. The cracking algorithms tested achieved minimal success until $10^7$ guesses, but succeeded against about 48% of accounts by $10^{14}$ guesses. Depending as they do on a single cracking algorithm, these must be considered the worst-case success rates for an attacker; it is quite possible that better tuning would greatly improve attack performance. In particular, it is not safe to assume that this policy ensures good survival

| Attack | | Guesses | Recommended defenses |
|---|---|---|---|
| Online guessing | Breadth-first | $10^4$ | Password blacklist; rate-limiting; account lock-out; recognition of known devices (e.g., by browser cookies, IP address recognition) |
| | Depth-first | $10^6$ | |
| Offline guessing | Breadth-first | $10^{14}$ | Iterated hashing; prevent leak of hashed-password file; keyed hash functions with Hardware Security Module support (Sections 3.4, 3.6) |
| | Depth-first | $10^{20}$ | |
| Rainbow table lookup (using extensive pre-computation) | | n/a | Salting; prevent leak of hashed-password file |
| Non-guessing (phishing, keylogging, network sniffing) | | n/a | Beyond scope of this paper |

Table 5: Selected attack types, number of per-account guesses expected in moderate attacks, and recommended defenses. We assume a 4-month guessing campaign, and for offline guessing that the password file is salted and hashed (see Section 3.4). Rate-limiting includes delays and various other techniques limiting login attempts over fixed time periods (see Section 4.4). Rainbow tables are explained in Section 3.2.

up to $10^7$ guesses, since most cracking algorithms optimize performance at high rather than low guess numbers.

In answering whether password policies work, we must first decide what it is we want of them. We use Section 3.3 which argued that safety from depth-first online guessing requires withstanding $10^6$ guesses, while safety from offline guessing requires $10^{14}$ or more. There are many tools that increase resistance to online guessing; some offer a simple way to protect against online guessing with lower usability impact than composition policies—e.g., password blacklists (see Section 4.2).

The above and further evidence suggest that composition policies are mediocre at protecting against offline guessing. For example, over 20% of CMU passwords were found in fewer than $10^{11}$ guesses, and 48% after $10^{14}$ [35]. While the stringent policy (minimum length eight and inclusion of all four character classes) has forced half of the population to cross the online-offline chasm, for practical purposes this is still failure: we expect most administrators would regard a site where half of the passwords are in an attacker's hands as being 100% compromised. Of policies studied by Kelley et al. [32] only one that required 16 characters gave over 80% survival rate at $10^{14}$ guesses.

Thus, the ubiquity of composition policies (which we expect stems from historical use, zero direct system cost, and the ease of giving advice) is at odds with a relatively modest delivery: they help protect against online attacks, but alternatives seem better. Some policies increase guess-resistance more than others, but none delivers robust resistance against the level of guessing modern offline attacks can bring to bear. Given that no aspect of password security seems to incite comparable user animosity [1, 14, 41], and that this is exacerbated by the rise of mobile devices with soft keyboards, composition policies appear to offer very poor return on user effort.

## 4.2 Blacklists and proactive checking

Another method to avoid weak passwords is to use a blacklist of known bad choices which are forbidden, sometimes called *proactive password checking* [5, 55]. This can be complementary or an alternative to a composition policy. Microsoft banned common choices for hotmail in 2011. In 2009, Twitter banned a list of 370 passwords, which account for (case insensitive) 5.2% of Rockyou accounts; simply blocking these popular passwords helps a significant fraction of users who would otherwise be at extreme risk of online guessing.

Also examining efficacy, using a blacklist of 50,000 words Weir et al. [60] found that over 99% of passwords withstood 4,000 guesses; 94% withstood 50,000. Thus, a simple blacklist apparently offers excellent protection against breadth-first online attacks and good improvement for depth-first online attacks.

Blacklists of a few thousand, even one million passwords, can be built by taking the commonest choices from leaked distributions. At $10^6$ they may offer excellent protection against all online attacks. However, they do not offer much protection against offline attacks. Blacklists of size $10^{14}$ appear impractical. A significant annoyance issue also increases with list size [35]: users may understand if a few thousand or even $10^6$ of the most common choices are forbidden, but a list of $10^{14}$ appears capricious and (in contrast to composition policies) it is not possible to give clear instructions on how to comply.

As an advantage of blacklists, they inconvenience only those most at risk. 100% of users using one of Twitter's 370 black-words is highly vulnerable to online guessing. By contrast, forcing compliance with a composition policy inconveniences all users (including those with long lowercase passwords that resist offline guessing quite well [35]) and apparently delivers little.

There is a risk that a static blacklist lacks currency; band names and song lyrics can cause popularity surges that go unrepresented—e.g., the 16 times that `justinbieber` appears in the 2009 Rockyou dataset would likely be higher in 2014. Also, even if the top $10^6$ passwords are banned, something else becomes the new most common password. The assumption is that banning the current most popular choices results in a distribution that is less skewed; this assumption does not seem strong, but has not been empirically verified. In one proposed password storage scheme that limits the popularity of any password [54], no more than $T$ users of a site are allowed to have the same password (for a configurable threshold $T$); this eliminates the currency problem and reduces the head-end password distribution skew.

### 4.3 Expiration policies (password aging)

Forced password change at regular intervals is another well-known recommendation, endorsed by NIST [13] and relatively common among enterprises and universities, albeit rarer among general web-sites. Of 75 sites examined in one study [23], 10 of 23 universities forced such a policy, while 4 of 10 government sites, 0 of 10 banks, and 0 of 27 general purpose sites did so.

The original justification for password aging was apparently to reduce the time an attacker had to guess a password. Expiration also limits the time that an attacker has to exploit an account. Ancillary benefits might be that it forces users into a different password selection strategy, e.g., if passwords expire every 90 days, it is less likely that users choose very popular problematic choices like `password` and `abcdefg` and more likely that they develop strategies for passwords that are more complex but which can be modified easily (e.g., incrementing a numeric substring). As a further potential benefit, it makes re-use between accounts less likely—whereas reusing a static password across accounts is easy and common [22], forced expiration imposes co-ordination overhead for passwords re-used across sites.

Reducing guessing time is relevant for offline attacks (an online guesser, as noted, gets far fewer attempts). So any benefit against guessing attacks is limited to cases where offline guessing is a factor, which Section 3.2 argues are far less common.

Reducing the time an attacker has to exploit an account is useful only if the original avenue of exploitation is closed, and no alternate (backdoor) access means has been installed. When the NIST guidelines were written, guessing was a principal means of getting a password. An attacker who had successfully guessed a password would be locked out by a password change; he would have to start guessing anew. Several factors suggest that this benefit is now diminished. First, offline password guessing is now only one avenue of attack; if the password is gained by keylogging-malware, a password change has little effect if the malware remains in place. Second, even if the attack is offline guessing, expiration turns out to be less effective than believed. Zhang et al. [63] recently found many new passwords very closely related to old after a forced reset; they were given access to expired passwords at UNC and allowed (under carefully controlled circumstances) to submit guesses for the new passwords. The results are startling: they guessed 17% of passwords in 5 tries or fewer, and 41% of accounts in under 3 seconds of offline attacking.

Thus, with forced expiration, new passwords appear to be highly predictable from old, and the gain is slight, for a policy competing with composition rules as most-hated by users. The benefits of forcing users to different strategies of choosing passwords, and making re-use harder may be more important. Given the severe usability burden, and associated support costs, expiration should probably be considered only for the top end of the high-consequence category.

### 4.4 Rate-limiting and lockout policies

A well-known approach to limiting the number of online attack guesses is to impose some kind of lockout policy—e.g., locking an account after three failed login attempts (or 10, for a more user-friendly tradeoff [12]). It might be locked for a certain period of time, or until the user takes an unlocking action (e.g., by phoning, or answering challenge questions). Locking for an hour after three failed attempts reduces the number of guesses an online attacker can make in a 4-month campaign to $3 \times 24 \times 365/3 = 8,760$ (cf. Section 3.3). A related approach increasingly delays the system response after a small number of failed logins—to 1s, 2s, 4s and so on. Bonneau and Preibusch [10] found that in practice, very few sites block logins even after 100 failed logins (though the sites they studied were predominantly in the low and medium consequence categories). Secret questions (Section 4.6), if used, must similarly be throttled.

The two main problems with lockout policies are the resulting usability burden, and the denial of service vulnerability created. Usability is clearly an issue given that users forget passwords a great deal. The denial of service vulnerability is that a fixed lockout policy allows an attacker to lock selected users out of the site. Incentives may mean that this represents a greater problem for some categories of sites than others. An online auction user might lockout a rival as a deadline approaches; someone interested in mayhem might lock all users of an online brokerage out during trading hours.

Throttling online guessing while avoiding intentional service lockouts, was explored by Pinkas and Sander [47] and extended by others [2, 59]. Login attempts can be restricted to devices a server has previously associated with successful logins for a given username, e.g., by browser cookies or IP address; login attempts from other devices (assumed to be potential online guessing machines) require both a password and a correctly-answered CAPTCHA. Through a clever protocol, legitimate users logging in from new devices see only a tiny fraction of CAPTCHAs (e.g., 5% of logins from a first-time device). The burden on online guessers is much larger, due to a vastly larger number of login attempts. The downside of this approach is CAPTCHA usability.

## 4.5 Password meter effectiveness

In addition to offering tips or advice on creating good passwords, many large sites employ password meters, purportedly measuring password strength, in an attempt to nudge users toward better passwords. They are generally implemented in Javascript in the browser, severely limiting the complexity of the strength-estimation algorithm implemented—e.g., downloading a very large dictionary to check against is problematic. Thus many meters use flawed measures (see Section 3.1) which correlate poorly with guessing resistance. This also produces many incongruities, e.g., classifying `Pa$$w0rd` as "very strong" and `gunpyo` as "weak". Of course, deficiencies in currently deployed meters do not necessarily imply that the general idea is flawed.

Among recent studies of the efficacy of meters, Ur et al. [58] examined the effect of various meters on 2,931 Mechanical Turk users, finding that significant increases in guessing-resistance were only achieved by very stringent meters. The presence of any meter did however provide some improvement even in resistance to online attacks (i.e., below $10^6$ guesses). De Carnavelet and Mannan [17] compare several password meters in common use and find enormous inconsistencies: passwords being classified as strong by one are termed weak by another. Egelman et al. [20] explore whether telling users how their password fares relative to others might have a greater effect than giving an absolute measure. Those who saw a meter tended to choose stronger passwords than those who didn't, but the type of meter did not make a significant difference. In a post-test survey 64% of participants admitted reusing a password from elsewhere—such users may have been influenced to re-use a different old password, but every old password is obviously beyond the reach of subsequent influences.

## 4.6 Backup questions & reset mechanisms

Reset mechanisms are essential at almost every password-protected site to handle forgotten passwords. For most cases, it can be assumed the user still has access to a secondary communication channel (e.g., an e-mail account or phone number on record)—and the assumed security of that channel can be leveraged to provide the reset mechanism. A common practice is to e-mail back to the user either a reset link or temporary password.

Sites that store passwords cleartext or reversibly encrypted can e-mail back that password itself if forgotten, but this exposes the password to third parties. Mannan et al. [34] propose to allow forgotten passwords to be restored securely; the server stores an encrypted copy of the password, with the decryption key known to a user recovery device (e.g., smartphone) but not the server.

Many sites use backup authentication questions (*secret questions*) instead of, or in conjunction with, emailing a reset link. The advantage of doing both is that an attacker gaining access to a user's e-mail account could gain access to any sites that e-mail reset links. Different categories of accounts (see Section 2) must approach this question differently. For high-consequence accounts, it seems that backup questions should be asked to further authenticate the user; for lower consequence accounts, the effort of setting up and typing backup questions must be taken into account.

When a secondary communication channel is unavailable (e.g., the site in question is the webmail provider itself, or a secondary communication channel was never set up, or is no longer available) backup questions are widely used. Unfortunately, plentiful evidence [49, 53] shows that typically in practice, the guessing-space of backup question answers is obviously too small, or involves questions whose answers can be looked up on the Internet for targeted or popular personalities. Several high-profile break-ins have exploited this fact.

Proposed authentication alternatives exist (e.g., [52]), but require more study. In summary, the implementation of password reset mechanisms is sensitive, fraught with dangers, and may require case-specific decisions.

## 4.7 Phishing

Guessing is but one means to get a password. Phishing rose to prominence around 2005 as a simple way to socially engineer users into divulging secrets. There are two varieties. Generic or scattershot attempts are generally delivered in large spam campaigns; *spear phishing* aims at specific individuals or organizations, possibly with target-specific lures to increase effectiveness.

Scattershot phishing generally exploits user confusion as to how to distinguish a legitimate web-site from a spoofed version [19]. The literature suggests many approaches to combat the problem, e.g., toolbars, tokens, two-factor schemes, user training. Few of these have enjoyed large-scale deployment. One that did, the SiteKey image to allow a user to verify a site, was found not to meet its design goals [51]: most users entered their password at a spoofed site even in the absence of the trust indicator. A toolbar indicator study reached a similarly pessimistic conclusion [62]. Equally, no evidence suggests any success from efforts to train users to tell good sites from bad simply by parsing the URL; the task itself is ill-defined [29]. In fact, much of the progress against scattershot phishing in recent years appears to have been by browser vendors, through better identification and blocking of phishing sites.

Spear phishing continues to be a major concern, especially for high-consequence sites. The March 2011 breach on RSA Security's SecurID hardware tokens was reportedly[8] such an attack. It is too early to say if approaches wherein administrators send periodic (defensive training) phishing emails to their own users leads to improved outcomes.

## 4.8 Re-using email address as username

Many sites (over 90% by one study [10]) encourage or force users to use an email address as username. This provides a point of contact (e.g., for password resets—or marketing), ensures unique usernames, and is memorable. However it also brings several security issues.

It encourages users (subconsciously or otherwise) to re-use the email password, thereby increasing the threats based on password re-use [16]. It can facilitate forms of phishing if users become habituated to entering their email passwords at low-value sites that users email addresses as usernames.

Re-using email addresses as usernames across sites also facilitates leaking information regarding registered users of those sites [50], although whether a given string is a valid username at a site can be extracted for non-email address usernames also [10, 11]. Preventing such leaks may be as much a privacy issue, as a security issue.

## 5 Discussion and implications

## 5.1 System-side vs. client-side defences

Some password-related defences involve implementation choices between system-side and client-side mechanisms; some attacks can be addressed at either the server (at cost of engineering effort) or the client (often at cost of user effort). Table 6 summarizes costs and benefits of several measures that we have discussed, noting security benefit and usability cost.

We have seen little discussion in the literature of the available trade-offs—and implications on cost, security, usability, and system-wide efficiency with respect to total user effort—between implementing password-related functionality client-side vs. server-side. Ideally, all decisions on where to impose costs would be made explicitly and acknowledged. A danger is that costs offloaded to the user are often hard to measure, and therefore unmeasured—this does not make the cost zero, but makes it hard to distinguish from zero. It is a natural consequence that system-side costs, which are more directly visible and more easily measured, are under-utilized, at the expense of client-side mechanisms which download (less visible, harder to measure) cognitive effort to end-users. For example, forcing users to choose passwords that will resist many guesses is a way of addressing the threat of offline attacks, and relies almost exclusively on user effort. Investing engineering time to better protect the password file, to ensure that leaks are likely to be detected, and to ensure that passwords are properly salted and hashed (or protected using an offline-resistant scheme such as discussed in Section 3.6) are alternatives dealing with the same problem that rely on server-side effort (engineering effort and/or operational time). Florêncio and Herley [23] found that sites where users do not have a choice (such as government and university sites) were more likely to address the offline threat with user effort, while sites that compete for users and traffic (such as retailers) were more likely to allow password policies that addressed the online threat only.

Scale is important in deciding how costs should be divided between the server and client sides; what is reasonable at one scale may be unacceptable at another. For example, many web-sites today have many more accounts than the largest systems of 30 years ago. A trade-off inconveniencing 200 users to save one systems administrator effort might be perfectly reasonable; however, the same trade-off involving 100 million users and 10 administrators is a very different proposition: the factor of $50,000$ increase in the ratio of users to administrators means that decisions should be approached differently, especially in any environment where user time, energy, and effort is a limited resource. There is evidence that the larger web-sites take greater care than smaller ones to reduce the burden placed on users [23].

## 5.2 Take-away points

We now summarize some of the key findings, and make recommendations based on the analysis above.

Many different types of sites impose passwords on users; asset values related to these sites and associated accounts range widely, including different valuations between users of the same sites. Thus, despite little attention to date in the literature, recognizing different categories of accounts is important (cf. Table 1). User effort available for managing password portfolios is finite [3, 25, 27, 57]. Users should spend less effort on password management issues (e.g., choosing complex passwords) for don't-care and lower consequence accounts, allowing more effort on higher consequence accounts. Password re-use across accounts in different categories is dangerous; a major concern is lower consequence sites compromising passwords re-used for high-consequence sites. While this seems an obvious concern, a first step is greater formal recognition of different categories of sites. We summarize this take-away point as:

T1: *Recognizing different categories of web-sites is essential to responsibly allocating user password*

[8] http://www.wired.com/2011/08/how-rsa-got-hacked/

| IMPLEMENTATION ASPECT | ATTACKS STOPPED OR SLOWED | USER IMPACT | REMARKS |
|---|---|---|---|
| Password stored non-plaintext | Full compromise on server breakin alone | None | Recommended |
| Salting (global and per-account) | Pre-computation attacks (table lookup) | None | Recommended |
| Iterated hashing | Slows offline guessing proportionally | None | Recommended |
| MAC of iterated, salted hash | Precludes offline guessing (requires key) | None | Best option (key management) |
| Rate-limiting & lockout policies | Hugely reduces online guessing | Possible user lockout | Recommended |
| Blacklisting (proactive checking) | Eliminates most-probable passwords | Minor for small lists | Recommended |
| Length rules | Slows down naive brute force attacks | Cognitive burden | Recommended: length $\geq 8$ |
| Password meters | Nudges users to "less guessable" passwords | Depends on user choice | Marginal gain |
| Password aging (expiration) | Limits ongoing attacker access; indirectly ameliorates password re-use | Significant; annoying | Possibly more harm than good |
| Character-set rules | May slow down naive brute-force attacks | Cognitive burden. Slows entry on mobile devices | Often bad return on user effort |

Table 6: Password-related implementation options. The majority of Remarks are relevant to medium-consequence accounts (see Table 1). It is strongly recommended that password storage details (e.g., salting, iterated hashing, MAC if used) are implemented by standard library tools.

*management effort across sites. Users are best served by effort spent on higher consequence sites, and avoiding cross-category password re-use.*

While naive "password strength" measures are widely used, simple to calculate, and have formed the basis for much of the analysis around passwords, simplistic metrics [13] based on Shannon entropy are poor measures of guessing-resistance (recall Section 3.1). Reasoning that uses naive metrics as a proxy for security is unsound and leads to unreliable conclusions. Policies, requirements and advice that seek to improve password security by "increasing entropy" should be disregarded.

T2: *Crude entropy-based estimates are unsuitable for measuring password resistance to guessing attacks; their use should be discouraged.*

While choosing passwords that will resist (online and/or offline) guessing has dominated the advice directed at users, it is worth emphasizing that the success rate of several attacks are unaffected by password choice.

T3: *The success of threats such as client-side malware, phishing, and sniffing unencrypted wireless links are entirely unaffected by password choice.*

Password policies and advice aim to have users choose passwords that will withstand guessing attacks. The threshold number of guesses to survive online and offline attacks differ enormously. The first threshold does not grow as hardware and cracking algorithms improve; the second gradually increases with time, only partially offset by adaptive password hashing functions (if used).

T4: *Password guessing attacks are either online or offline. The guessing-resistance needed to survive the two differs enormously. Withstanding $10^6$ guesses probably suffices for online; withstanding $10^{14}$ or more guesses may be needed to resist determined, well-resourced offline attacks.*

There is no continuum of guessing attack types—it is either online or offline, with nothing in between. There is a chasm between the threshold to withstand these two different types. There is little security benefit in exceeding the online threshold while failing to reach the offline one. Passwords that fail to completely cross this chasm waste effort since they do more than is necessary to withstand online attacks, but still succumb to offline attacks.

T5: *Between the thresholds to resist online and offline attacks, incremental improvement in guess-resistance has little benefit.*

Recall that rainbow table attacks are one form of offline attack, and require access to leaked password hashes.

T6: *Rainbow table attacks can be effectively stopped by well-known salting methods, or by preventing the leakage of hashed password files.*

Analysis of Fig.1 shows that offline attacks are possible and necessary in only very limited circumstances which occur far less often than suggested from the attention given by the research literature. If the password file has not been properly salted and hashed, then user effort to withstand beyond $10^6$ guesses is better spent elsewhere.

T7: *Offline guessing attacks are a major concern only if the password file leaks, the leak goes undetected, and the file was properly salted and hashed (otherwise simpler attacks work, e.g., rainbow tables).*

It follows that sites that store passwords in plaintext or reversibly encrypted, and impose strict password composition policies unnecessarily burden users—the policies offer zero benefit against intelligent attackers, as any increased guessing-resistance is irrelevant. The attacker either has direct access to a plaintext password, or if the key encrypting the hashed password does not also leak then the (plaintext) password hashes needed for the offline guessing attack are unavailable.

T8: *For implementations with stored passwords available at the server (plaintext or reversibly encrypted), composition policies aiming to force resistance to offline guessing attacks are unjustifiable—no risk of offline guessing exists.*

The threat of offline guessing attacks can essentially be eliminated if it can be ensured that password files do not leak, e.g., by keyed hash functions with HSM (hardware security) support. Guessing attack risks then reduce to online guessing, which is addressable by known mechanisms such as throttling, recognizing known devices, and proactive checking to disallow too-popular passwords—all burdening users less than composition policies.

T9: *Online attacks are a fact of life for public-facing servers. Offline attacks, by contrast, can be entirely avoided by ensuring the password file does not leak, or mitigated by detecting if it does leak and having a disaster-recovery plan to force a system-wide password reset in that case.*

## 6 Concluding remarks

In concluding we summarize the case against consuming user effort in attempts to resist offline guessing attacks.

1. Honesty demands a clear acknowledgement that we don't know how to do so: attempts to get users to choose passwords that will resist offline guessing, e.g., by composition policies, advice and strength meters, must largely be judged failures. Such measures may get *some* users across the online-offline chasm, but this helps little unless it is a critical mass; we assume most administrators would consider a site with half its passwords in an attacker's hands to be fully rather than half compromised.

2. Failed attempts ensure a large-scale waste of user effort, since exceeding the online while falling short of the offline threshold delivers no security benefit.

3. The task gets harder every year—hardware advances help attackers more than defenders, increasing the number of guesses in offline attacks.

4. Zero-user-burden mechanisms largely or entirely eliminating offline attacks exist, but are little-used.

5. Demanding passwords that will withstand offline attack is a defense-in-depth approach necessary only when a site has failed both to protect the password file, and to detect the leak and respond suitably.

6. That large providers (e.g., Facebook, Fidelity, Amazon) allow 6-digit PINs demonstrates that it is possible to run first-tier properties without placing the burden of resisting offline attacks on users.

Preventing, detecting and recovering from offline attacks must be administrative priorities, if the burden is not to be met with user effort. It is of prime importance to ensure that password files do not leak (or have content such that leaks are harmless), that any leak can be quickly detected, and that an incident response plan allows system-wide forced password resets if and when needed. Next, and of arguably equal importance, is protecting against online attacks by limiting the number of online guesses that can be made (e.g., by throttling or lockouts) and precluding the most common passwords (e.g., by password blacklists). Salting and iterated hashing are of course expected, using standardized adaptive password hashing functions or related MACs.

**Acknowledgements.** We thank Michael Brogan and Nathan Dors (U. Washington) for helpful discussions, anonymous referees, and Furkan Alaca, Lujo Bauer, Kemal Bicakci, Joseph Bonneau, Bill Burr, Nicolas Christin, Simson Garfinkel, Peter Gutmann, M. Mannan, Fabian Monrose, and Julie Thorpe for detailed comments on a draft. The third author acknowledges an NSERC Discovery Grant and Canada Research Chair in Authentication and Computer Security.

## References

[1] A. Adams and M. A. Sasse. Users Are Not the Enemy. C.ACM, 42(12), 1999.

[2] M. Alsaleh, M. Mannan, and P. C. van Oorschot. Revisiting defenses against large-scale online password guessing attacks. IEEE TDSC, 9(1):128–141, 2012.

[3] A. Beautement and A. Sasse. The economics of user effort in information security. Computer Fraud & Security, pages 8–12, October 2009.

[4] A. Beautement, M. Sasse, and M. Wonham. The Compliance Budget: Managing Security Behaviour in Organisations. In NSPW, 2008.

[5] F. Bergadano, B. Crispo, and G. Ruffo. High dictionary compression for proactive password checking. ACM Trans. Inf. Syst. Secur., 1(1):3–25, 1998.

[6] J. Bonneau. Guessing human-chosen secrets. University of Cambridge. Ph.D. thesis, May 2012.

[7] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In Proc. IEEE Symp. on Security and Privacy, pages 538–552, 2012.

[8] J. Bonneau. Password cracking, part II: when does password cracking matter, Sept.4, 2012. `https://www.lightbluetouchpaper.org`.

[9] J. Bonneau, C. Herley, P. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In Proc. IEEE Symp. on Security and Privacy, 2012.

[10] J. Bonneau and S. Preibusch. The password thicket: Technical and market failures in human authentication on the web. In WEIS, 2010.

[11] A. Bortz and D. Boneh. Exposing private information by timing web applications. Proc. WWW, 2007.

[12] S. Brostoff and M. Sasse. "Ten strikes and you're out": Increasing the number of login attempts can improve password usability. CHI Workshop, 2003.

[13] W. Burr, D. F. Dodson, and W. Polk. Electronic Authentication Guideline. In NIST Special Pub 800-63, 2006.

[14] W. Cheswick. Rethinking passwords. ACM Queue, 10(12):50–56, 2012.

[15] G. D. Crescenzo, R. J. Lipton, and S. Walfish. Perfectly secure password protocols in the bounded retrieval model. In TCC, pages 225–244, 2006.

[16] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. NDSS, 2014.

[17] X. de Carnavalet and M. Mannan. From very weak to very strong: Analyzing password-strength meters. In Proc. NDSS, 2014.

[18] S. Designer and S. Marechal. Password Security: Past, Present, Future (with strong bias towards password hashing), December 2012. Slide deck: `http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/`.

[19] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In Proc. CHI, 2006.

[20] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven? the impact of password meters on password selection. In Proc. CHI, 2013.

[21] S. Fahl, M. Harbach, Y. Acar, and M. Smith. On the ecological validity of a password study. In Proc. SOUPS. ACM, 2013.

[22] D. Florêncio and C. Herley. A Large-Scale Study of Web Password Habits. Proc. WWW, 2007.

[23] D. Florêncio and C. Herley. Where Do Security Policies Come From? Proc. SOUPS, 2010.

[24] D. Florêncio, C. Herley, and B. Coskun. Do Strong Web Passwords Accomplish Anything? Proc. Usenix Hot Topics in Security, 2007.

[25] D. Florêncio, C. Herley, and P. van Oorschot. Password portfolios and the finite-effort user: Sustainably managing large numbers of accounts. In Proc. USENIX Security, 2014.

[26] D. Goodin. Why passwords have never been weaker and crackers have never been stronger, 2012. Ars Technia, `http://arstechnica.com/security/2012/08/passwords-under-assault/`.

[27] B. Grawemeyer and H. Johnson. Using and managing multiple passwords: A week to a view. Interacting with Computers, 23(3):256–267, 2011.

[28] E. Grosse and M. Upadhyay. Authentication at scale. IEEE Security & Privacy, 11(1):15–22, 2013.

[29] C. Herley. So Long, And No Thanks for the Externalities: Rational Rejection of Security Advice by Users. Proc. NSPW, 2009.

[30] C. Herley and P. van Oorschot. A research agenda acknowledging the persistence of passwords. IEEE Security & Privacy, 10(1):28–36, 2012.

[31] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In Proc. ACM CCS, pages 145–160, 2013.

[32] 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 Proc. IEEE Symp. on Security and Privacy, 2012.

[33] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. Proc. ISW'97—Springer LNCS, 1396:121-134, 1998.

[34] M. Mannan, D. Barrera, C. D. Brown, D. Lie, and P. C. van Oorschot. Mercury: Recovering forgotten passwords using personal devices. In Financial Cryptography, pages 315–330, 2011.

[35] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In ACM CCS, 2013.

[36] A. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[37] A. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[38] R. Morris and K. Thompson. Password Security: A Case History. C.ACM, 22(11):594–597, 1979.

[39] A. Muller, M. Meucci, E. Keary, and D. Cuthbert, editors. OWASP Testing Guide 4.0. Section 4.5: Authentication Testing (accessed July 27, 2014), `https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents`.

[40] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. ACM CCS, 2005.

[41] D. Norman. The Way I See It: When security gets in the way. Interactions, 16(6):60–63, 2009.

[42] oclHashcat. `http://www.hashcat.net/`.

[43] P. Oechslin. Making a faster cryptanalytical time-memory trade-off. Advances in Cryptology - CRYPTO 2003, 2003.

[44] Openwall. `http://www.openwall.com/john/`.

[45] OWASP. Guide to Authentication. Accessed July 27, 2014, `https://www.owasp.org/index.php/Guide_to_Authentication`.

[46] C. Percival. Stronger key derivation via sequential memory-hard functions. In BSDCan, 2009.

[47] B. Pinkas and T. Sander. Securing Passwords Against Dictionary Attacks. ACM CCS, 2002.

[48] N. Provos and D. Mazieres. A future-adaptable password scheme. In USENIX Annual Technical Conference, FREENIX Track, pages 81–91, 1999.

[49] R. W. Reeder and S. Schechter. When the password doesn't work: secondary authentication for websites. IEEE Security & Privacy, 9(2):43–49, 2011.

[50] P. F. Roberts. Leaky web sites provide trail of clues about corporate executives. ITworld.com, August 13, 2012.

[51] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators: evaluation of website authentication and effect of role playing on usability studies. In Proc. IEEE Symp. on Security and Privacy, 2007.

[52] S. Schechter, S. Egelman, and R. Reeder. It's not what you know, but who you know: a social approach to last-resort authentication. In Proc. CHI, 2009.

[53] S. E. Schechter, A. J. B. Brush, and S. Egelman. It's no secret: Measuring the security and reliability of authentication via "secret" questions. In Proc. IEEE Symp. Security & Privacy, 2009.

[54] Schechter, S. and Herley, C. and Mitzenmacher, M. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. Proc. HotSec, 2010.

[55] E. H. Spafford. OPUS: Preventing weak password choices. Computers & Security, 11(3):273–278, 1992.

[56] J. Steven and J. Manico. Password Storage Cheat Sheet (OWASP). OWASP. Apr.7, 2014, `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`.

[57] E. Stobert and R. Biddle. The password life cycle: user behaviour in managing passwords. In Proc. SOUPS, 2014.

[58] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, et al. How does your password measure up? The effect of strength meters on password creation. In Proc. USENIX Security, 2012.

[59] P. van Oorschot and S. Stubblebine. On Countering Online Dictionary Attacks with Login Histories and Humans-in-the-Loop. ACM TISSEC, 9(3):235–258, 2006.

[60] M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In Proc. ACM CCS, 2010.

[61] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In Proc. IEEE Symp. on Security and Privacy, pages 391–405, 2009.

[62] M. Wu, R. Miller, and S. L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks. Proc. CHI, 2006.

[63] Y. Zhang, F. Monrose, and M. K. Reiter. The security of modern password expiration: An algorithmic framework and empirical analysis. In Proc. ACM CCS, 2010.

[64] E. Zwicky. Brute force and ignorance. ;login:, 35(2):51–52, April 2010. USENIX.

# Analyzing Log Analysis: An Empirical Study of User Log Mining

S. Alspaugh[*] *UC Berkeley*  Beidi Chen *UC Berkeley*  Jessica Lin *UC Berkeley*

Archana Ganapathi *Splunk Inc.*  Marti A. Hearst *UC Berkeley*  Randy Katz *UC Berkeley*

## Abstract

We present an in-depth study of over 200K log analysis queries from Splunk, a platform for data analytics. Using these queries, we quantitatively describe log analysis behavior to inform the design of analysis tools. This study includes state machine based descriptions of typical log analysis pipelines, cluster analysis of the most common transformation types, and survey data about Splunk user roles, use cases, and skill sets. We find that log analysis primarily involves filtering, reformatting, and summarizing data and that non-technical users increasingly need data from logs to drive their decision making. We conclude with a number of suggestions for future research.

**Tags:** log analysis, query logs, user modeling, Splunk, user surveys

## 1 Introduction

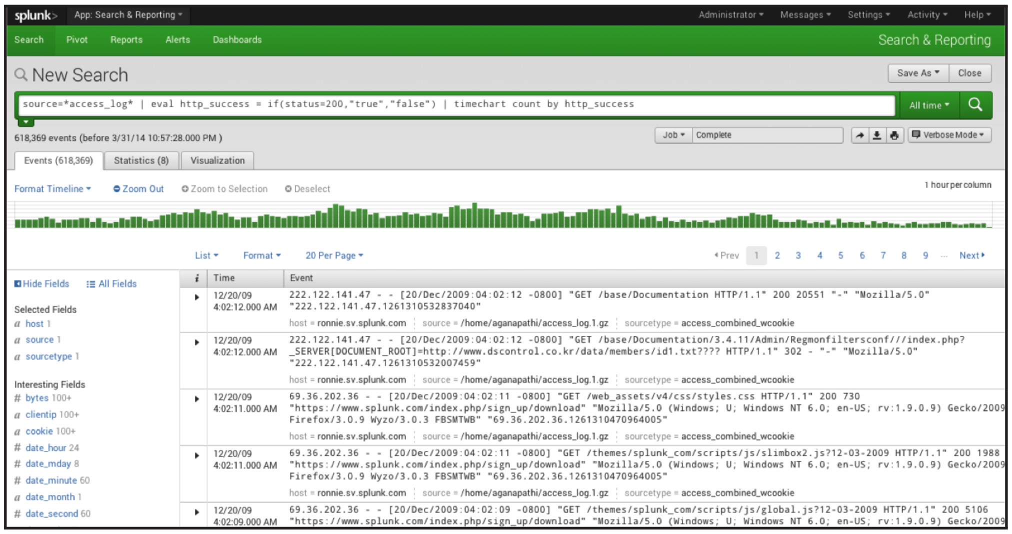
Log analysis is the process of transforming raw log data into information for solving problems. The market for log analysis software is huge and growing as more business insights are obtained from logs. Stakeholders in this industry need detailed, quantitative data about the log analysis process to identify inefficiencies, streamline workflows, automate tasks, design high-level analysis languages, and spot outstanding challenges. For these purposes, it is important to understand log analysis in terms of discrete tasks and data transformations that can be measured, quantified, correlated, and automated, rather than qualitative descriptions and experience alone.

This paper helps meet this need using over 200K queries recorded from a commercial data analytics system called Splunk. One challenge is that logged system events are not an ideal representation of human log analysis activity [3]. Logging code is typically not designed to capture human behavior at the most efficacious level of granularity. Even if it were, recorded events may not reflect internal mental activities. To help address this gap, we supplement the reported data with results of a survey of Splunk sales engineers regarding how Splunk is used in practice.

In our analysis, we examine questions such as: What transformations do users apply to log data in order to analyze it? What are common analysis workflows, as described by sequences of such transformations? What do such workflows tell us, qualitatively, about the nature of log analysis? Who performs log analysis and to what end? What improvements do we need to make to analysis tools, as well as to the infrastructure that logs activities from such tools, in order to improve our understanding of the analysis process and make it easier for users to extract insights from their data?

The answers to these questions support a picture of log analysis primarily as a task of filtering, reformatting, and summarizing. Much of this activity appears to be data munging, supporting other reports in the literature [28]. In addition, we learn from our survey results that users outside of IT departments, including marketers and executives, are starting to turn to log analysis to gain business insights. Together, our experience analyzing these queries and the results of our analysis suggest several important avenues for future research: improving data transformation representation in analytics tools, implementing integrated provenance collection for user activity record, improving data analytics interfaces and creating intelligent predictive assistants, and further analyzing other data analysis activities from other systems and other types of data besides logs.

[*]This author was an employee of Splunk Inc. when this paper was written.



```
source=*access_log* | eval http_success = if(status=200,"true","false") | timechart count by http_success
```

618,369 events (before 3/31/14 10:57:28.000 PM )

Selected Fields
a host 1
a source 1
a sourcetype 1

Interesting Fields
# bytes 100+
a clientip 100+
a cookie 100+
# date_hour 24
# date_mday 8
# date_minute 60
a date_month 1
# date_second 60

```
12/20/09 4:02:12.000 AM
222.122.141.47 - - [20/Dec/2009:04:02:12 -0800] "GET /base/Documentation HTTP/1.1" 200 20551 "-" "Mozilla/5.0"
"222.122.141.47.1261310532837040"
host = ronnie.sv.splunk.com | source = /home/aganapathi/access_log.1.gz | sourcetype = access_combined_wcookie

12/20/09 4:02:12.000 AM
222.122.141.47 - - [20/Dec/2009:04:02:12 -0800] "GET /base/Documentation/3.4.11/Admin/Regmonfiltersconf///index.php?
_SERVER[DOCUMENT_ROOT]=http://www.dscontrol.co.kr/data/members/id1.txt???? HTTP/1.1" 302 - "-" "Mozilla/5.0"
"222.122.141.47.1261310532007459"
host = ronnie.sv.splunk.com | source = /home/aganapathi/access_log.1.gz | sourcetype = access_combined_wcookie

12/20/09 4:02:11.000 AM
69.36.202.36 - - [20/Dec/2009:04:02:11 -0800] "GET /web_assets/v4/css/styles.css HTTP/1.1" 200 730
"https://www.splunk.com/index.php/sign_up/download" "Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.9) Gecko/2009
Firefox/3.0.9 Wyzo/3.0.3 FBSMTWB" "69.36.202.36.1261310470964005"
host = ronnie.sv.splunk.com | source = /home/aganapathi/access_log.1.gz | sourcetype = access_combined_wcookie

12/20/09 4:02:11.000 AM
69.36.202.36 - - [20/Dec/2009:04:02:11 -0800] "GET /themes/splunk_com/scripts/js/slimbox2.js?12-03-2009 HTTP/1.1" 200 1988
"https://www.splunk.com/index.php/sign_up/download" "Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.9) Gecko/2009
Firefox/3.0.9 Wyzo/3.0.3 FBSMTWB" "69.36.202.36.1261310470964005"
host = ronnie.sv.splunk.com | source = /home/aganapathi/access_log.1.gz | sourcetype = access_combined_wcookie

12/20/09 4:02:09.000 AM
69.36.202.36 - - [20/Dec/2009:04:02:09 -0800] "GET /themes/splunk_com/scripts/js/global.js?12-03-2009 HTTP/1.1" 200 5106
"https://www.splunk.com/index.php/sign_up/download" "Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.9) Gecko/2009
```

Figure 1: The default Splunk GUI view displays the first several events indexed, with extracted fields highlighted on the side, and a histogram of the number of events over time displayed along the top. The user types their query into the search bar at the top of this view.

## 2 Related Work

We discuss (1) systems for log analysis, (2) techniques for log analysis, and (3) results of log analysis, so that those log analysis activities can be compared to our observations. We also discuss (4) user studies of system administrators – one of the primary classes of log analysts – and (5) of search engine users – where query logs are the main source of data on user behavior and needs.

**Systems for log analysis:** The purpose of this section is not to compare Splunk to other analysis systems, but to describe the uses these systems support, to provide a sense of how our observations fit within the larger context. Dapper, Google's system tracing infrastructure, is used by engineers to track request latency, guarantee data correctness, assess data access costs, and find bugs [34]. From their detailed descriptions, we can infer that engineers use transformations similar to those used by Splunk users. Other systems, such as Sawzall and PigLatin, include query languages that extract data from logs with heavy use of these same types of transformations [30, 26]. These points suggest that the activity records we have collected may represent typical log analysis usage, despite being gathered from only one system.

**Techniques for log analysis:** Published techniques for log analysis center around the main challenges in working with logs, such as dealing with messy formats, and solving event-based problems [23]. This includes event and host clustering [20, 21], root failure diagnosis [8, 17], anomaly detection [18], dependency inference [25, 19], and data extraction [16, 39]. Although their motivating use cases overlap with Splunk use cases, in our observations, the use of such techniques appears to be relatively rare (even though Splunk does provide, e.g., clustering and anomaly detection functionality).

**Results of log analysis:** Log analysis is also used in research as a means to an end rather than as the subject itself. Logs have been used to explain system behavior [7, 6], understand failures [31, 24], identify design flaws [11], spot security vulnerabilities [15], highlight new phenomena [29], and drive system simulations [12]. To the extent that such research involves heavy application of human inference rather than "automatic" statistical inference techniques, like many of those mentioned in the previous section, it appears to more closely align with our observations of log analysis behavior. However, the problems addressed are naturally of an academic nature, whereas Splunk users of often looking for timely business insights specific to their situation.

**System administrator user studies:** As system administrators are one of the primary classes of log analysts, studies of their behavior are relevant to our study of log analysis. Researchers have studied system administrators to characterize their work environments and problems commonly faced [4], as well as the mental models they form [13]. One study surveying 125 system administrators discovered that accuracy, reliability, and credibility are considered the most important features in tools [38]. Other researchers have called for more standardization in system administration activities – such efforts will benefit from the data we present [9].

| *Term* | *Definition* |
|---|---|
| event | a raw, timestamped item of data indexed by Splunk, similar to a tuple or row in databases |
| field | a key corresponding to a value in an event, similar to the concept of a column name |
| value | part of an event corresponding to a certain field, similar to a particular column entry in a particular row |
| query | a small program written in the Splunk query language, consisting of pipelined stages |
| stage | a portion of a query syntactically between pipes; conceptually a single transformation |
| transformation | an abstract category of similar commands e.g., filter or aggregate; each stage is a transformation |
| command | the part of a stage that indicates what operation to apply to the data |
| argument | the parts of a stage that indicate what fields, values, or option values to use with a command |
| interactive | a query that is run when it is entered by the user into the search bar |
| scheduled | a query that has been saved by a user and scheduled to run periodically like a `cron` job |

Table 1: Terminology describing Splunk data.

**Search engine query log studies:** While we are unaware of prior work that uses query logs to study *analysis* behavior, query logs are often used to study *search engine user* behavior. People have used search engine query logs to model semantic relationships [22], track user preferences [35], and identify information needs [32]. Techniques involve examining query terms and analyzing user sessions [14, 33]. Due to data quality issues discussed in Section 4, we could not analyze user sessions, but other aspects of our current and previous work parallel these techniques [2]. Employing some of these techniques to examine data analysis activity logs is a promising avenue of future research. Going forward we expect that the study of human information seeking behavior will be enriched through the study of analysis query logs.

## 3 Splunk Logs and Queries

We collected queries from Splunk[1], a platform for indexing and analyzing large quantities of data from heterogeneous data sources, especially machine-generated logs. Splunk is used for a variety of data analysis needs, including root cause failure detection, web analytics, A/B testing and product usage statistics. Consequently, the types of data sets indexed in Splunk also span a wide range, such as system event logs, web access logs, customer records, call detail records, and product usage logs. This section describes the Splunk data collection and query language in more detail; Table 1 lists the terminology introduced in this section.

### 3.1 Overview

**Data collection** To use Splunk, the user indicates the data that Splunk must index, such as a log directory on a file system. Splunk organizes this data into temporal *events* by using timestamps as delineators, and processes these events using a MapReduce-like architecture [5]. Splunk does not require the user to specify a schema for the data, because much log data is semi-structured or unstructured, and there is often no notion of a schema that can be imposed on the data a priori. Rather, *fields* and *values* are extracted from events at run time based on the *source type*. Specifically, when a user defines a new source type, Splunk guides the user in constructing regular expressions to extract fields and values from each incoming raw event.

[1] `www.splunk.com`

**Query llanguage** Splunk includes a query language for searching and manipulating data and a graphical user interface (GUI) with tools for visualizing query results. The query consists of a set of *stages* separated by the pipe character, and each stage in turn consists of a *command* and *arguments*. Splunk passes events through each stage of a query. Each stage filters, transforms or enriches data it receives from the previous stage, and pipes it to the subsequent stage, updating the displayed results as they are processed. A simple example of a query is a plain text search for specific strings or matching field-value pairs. A more complex example can perform more advanced transformations, such as clustering the data using k-means. Users can save certain queries and schedule them to be run on a given schedule, much like a `cron` job. We call these queries *scheduled* queries.

**Graphical user interface** Users almost always compose Splunk queries in the GUI. The default GUI view displays the first several events indexed, with extracted fields highlighted on the left hand side, and a histogram of the number of events over time displayed along the top. A screen shot of this default view is shown in Figure 1. The user types their query into the search bar at the top of this view. When the user composes their query in the GUI, we call it an *interactive* query.

When the user enters a query that performs a filter, the GUI updates to display events which pass through the filter. When the user uses a query to add or transform a field, the GUI displays events in updated form. Most queries result in visualizations such as tables, time series, and histograms, some of which appear in the GUI when the query is executed, in the "Visualization" tab (Figure 1). Users can also create "apps," which are custom views that display the results of pre-specified queries, possibly in real time, which is useful for things like monitoring and reporting. Although the set of visualizations Splunk offers does not represent the full breadth of all possible visualizations, they still capture a large set of standard, commonly used ones.

## 3.2 An Example Splunk Query

The Splunk query language is modeled after the Unix `grep` command and pipe operator. Below is an example query that provides a count of errors by detailed status code:

```
search error | stats count by status | lookup
    statuscodes status OUTPUT statusdesc
```

This example has three stages: `search`, `stats`, and `lookup` are the commands in each stage, `count by` and `OUTPUT` are functions and option flags passed to these commands, and "error", "status", "statuscodes", and "statusdesc" are arguments. In particular, "status" and "statusdesc" are fields.

To see how this query operates, consider the following toy data set:

| | | | |
|---|---|---|---|
| 0.0 | - | **error** | 404 |
| 0.5 | - | OK | 200 |
| 0.7 | - | **error** | 500 |
| 1.5 | - | OK | 200 |

The first stage of the query (`search` error) filters out all events not containing the word "error". After this stage, the data looks like:

| | | | |
|---|---|---|---|
| 0.0 | - | **error** | 404 |
| 0.7 | - | **error** | 500 |

The second stage (`stats count by` status) aggregates events by applying the `count` function over events grouped according to the "status" field, to produce the number of events in each "status" group.

| count | status |
|---|---|
| 1 | 404 |
| 1 | 500 |

The final stage (`lookup` status codes status `OUTPUT` statusdesc) performs a join on the "status" field between the data and an outside table that contains descriptions of each of the codes in the "status" field, and puts the corresponding descriptions into the "statusdesc" field.

| count | status | statusdesc |
|---|---|---|
| 1 | 404 | Not Found |
| 1 | 500 | Internal Server Error |

| | |
|---|---|
| Total queries | 203691 |
| Interactive queries | 18872 |
| Scheduled queries | 184819 |
| Distinct scheduled queries | 17085 |

Table 2: Characteristics of the set of queries analyzed from the Splunk logs.

## 4 Study Data

We collected over 200K Splunk queries. The data set consists of a list of timestamped query strings. Table 2 summarizes some basic information about this query set.

We wrote a parser for this query language; the parser is freely available [2]. This parser is capable of parsing over 90% of all queries in the data set, some of which may be valid failures, as the queries may be malformed. (This limitation only affects the cluster analysis in Section 6.)

It is important to note that we do not have access to any information about the data over which the queries were issued because these data sets are proprietary and thus unavailable. Having access only to query logs is a common occurrence for data analysis, and methodologies that can work under these circumstances are therefore important to develop. Further, by manually inspecting the queries and using them to partially reconstruct some data sets using the fields and values mentioned in the queries, we are fairly certain that these queries were issued over many different sources of data (e.g., web server logs, security logs, retail transaction logs, etc.), suggesting the results presented here will generalize across different datasets.

It is also important to note that some of the queries labeled as interactive in our data set turned out to be programmatically issued from sources external to Splunk, such as a user-written script. It is difficult to separate these mislabeled queries from the true interactive queries, so we leave their analysis to future work, and instead focus our analysis in this paper on scheduled queries.

[2] `https://github.com/salspaugh/splparser`

## 5 Transformation Analysis

The Splunk query language is complex and supports a wide range of functionality, including but not limited to: reformatting, grouping and aggregating, filtering, reordering, converting numerical values, and applying data mining techniques like clustering, anomaly detection, and prediction. It has 134 distinct core commands at the time of this writing, and commands are often added with each new release. In addition, users and Splunk app developers can define their own commands.

We originally attempted to analyze the logs in terms of command frequencies, but it was difficult to generalize from these in a way that is meaningful outside of Splunk [1]. So, to allow for comparisons to other log analysis workflows and abstract our observations beyond the Splunk search language, we manually classified these 134 commands into 17 categories representing the types of transformations encoded, such as filtering, aggregating, and reordering (Table 3).

Note that because some Splunk commands are overloaded with functionality, several commands actually perform multiple types of transformations, such as aggregation followed by renaming. In these cases, we categorized the command according to its dominant use case.

We use this categorization scheme to answer the following questions about log analysis activity:

- How are the individual data transformations statistically distributed? What are the most common transformations users perform? What are the least common?
- How are sequences of transformations statistically distributed? What type of transformations do queries usually start with? What do they end with? What transformations typically follow a given other transformation?
- How many transformations do users typically apply in a given query? What are the longest common subsequences of transformations?

### 5.1 Transformation Frequencies

We first counted the number of times that each transformation was used (Figure 2). The most common are **Cache** (27% of stages), **Filter** (26% of stages), **Aggregate** (10% of stages), **Macro** (10% of stages),and **Augment** (9% of stages). Scheduled queries are crafted and set up to run periodically, so the heavy use of caching and macros is unsurprising: Splunk adds caching to scheduled queries to speed their execution, and macros capture common workflows, which are likely to be discovered by users after the iterative, ad hoc querying that results in a “production-ready” scheduled query. Although we do not report directly on them here due to data quality issues (Section 4), anecdotally, it appears that interactive queries have a similar distribution except that the use of **Cache** and **Macro** is less frequent, and the use of **Input** is more frequent.

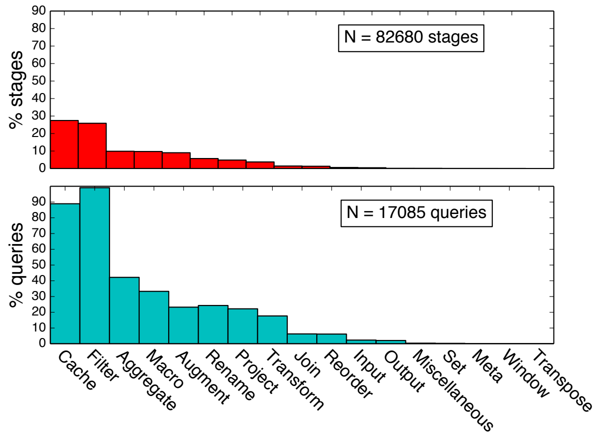


Figure 2: The distribution of data transformations that are used in log analysis. The top graph shows, for each transformation, the percent of stages that apply that transformation. The bottom graph shows, for each transformation, the percent of queries that contain that transformation at least once (so the percents do not add to 100).

For each transformation type, we also computed the number of queries that used that transformation (Figure 2). This gives us some idea of how many of the queries would be expressible in a restricted subset of the language, which is interesting because it tells us the relative importance of various transformations.

From this we see that **Filter** transformations are extremely important – 99% of scheduled queries use such transformations. Without **Aggregate** transformations, 42% of scheduled queries would not be possible. Around a quarter of queries use **Augment**, **Rename**, and **Project** transformations, and 17% use commands that **Transform** columns.

In contrast, **Join**s are only used in 6% of scheduled queries. This possible difference from database workloads could be because log data is not usually relational and generally has no schema, so it may often not have information that would satisfy key constraints needed for join, or it may already be sufficiently denormalized for most uses. It could also be because these are scheduled queries, and expensive **Join** operations have been optimized away, although again anecdotally the interactive queries do not suggest this. **Reorder** transformations are also used only 6% of the time – log events are already ordered by time by Splunk, and this is probably often the desired order. **Input** and **Output** transformations are used in only 2% of scheduled queries – these

| *Transformation* | *Description* | *Top Commands* | *% Queries* | *Examples* |
|---|---|---|---|---|
| Aggregate | coalesce values of a given field or fields (columns) into one summary value | `stats`<br>`timechart`<br>`top` | 86.0<br>9.0<br>3.0 | `stats sum(size_kb)`<br>`timechart count by region`<br>`top hostname` |
| Augment | add a field (column) to each event, usually a function of other fields | `eval`<br>`appendcols`<br>`rex` | 57.0<br>19.0<br>15.0 | `eval pct=count/total*100`<br>`spath input=json`<br>`rex "To:  (?<to>.*)"` |
| Cache | write to or read from cache for fast processing | `summaryindex`<br>`sitimechart` | 98.0<br>30.0 | `summaryindex namespace=foo`<br>`sitimechart count by city` |
| Filter | remove events (rows) not meeting the given criteria | `search`<br>`where`<br>`dedup` | 100.0<br>7.0<br>4.0 | `search name="alspaugh"`<br>`where count > 10`<br>`dedup session_id` |
| Input | input events into the system from elsewhere | `inputlookup` | 88.0 | `inputlookup data.csv` |
| Join | join two sets of events based on matching criteria | `join`<br>`lookup` | 82.0<br>16.0 | `join type=outer ID`<br>`lookup` |
| Macro | apply user-defined sequence of Splunk commands | `‘sourcetype_metrics‘`<br>`‘forwarder_metrics‘` | 50.0<br>13.0 | `‘sourcetype_metrics‘`<br>`‘forwarder_metrics‘` |
| Meta | configure execution environment | `localop` | 83.0 | `localop` |
| Miscellaneous | commands that do not fit into other categories | `noop` | 39.0 | `noop` |
| Output | write results to external storage or send over network | `outputlookup` | | `outputlookup results.csv` |
| Project | remove all columns except those selected | `table`<br>`fields` | 80.0<br>22.0 | `table region total`<br>`fields count` |
| Rename | rename fields | `rename` | 100.0 | `rename cnt AS Count` |
| Reorder | reorder events based on some criteria | `sort` | 100.0 | `sort - count` |
| Set | perform set operations on data | `append`<br>`set` | 66.0<br>40.0 | `append [...]`<br>`set intersect [...]  [...]` |
| Transform | mutate the value of a given field for each event | `fillnull`<br>`convert` | 96.0<br>2.0 | `fillnull status`<br>`convert num(run_time)` |
| Transpose | swap events (rows) with fields (columns) | `transpose` | 100.0 | `transpose` |
| Window | add fields that are windowing functions of other data | `streamstats` | 90.0 | `streamstats first(edge)` |

Table 3: Manual classification of commands in the Splunk Processing Language into abstract transformations categories. For each transformation category, the *Top Commands* column shows the most-used commands in that category. The *% Queries* column shows, for all queries containing a given transformation, what percent of queries contained that command.

again could have been optimized away, or possibly captured in **Macro**s. Lastly, the other transformations are used in nearly zero queries. In the case of **Window**ing transformations, this could be because windowed operations are accomplished "manually" through sequences of **Augment** transformations or via overloaded commands that were classified as other transformation types. We were surprised such operations were not more common. In the case of the others, such as **Transpose**, it is more likely because log data is rarely of the type for which such operations are needed.

## 5.2 Transformation Pipelines

Next, for each pair of transformation types, we counted the number of times within a query that the first transformation of the pair was followed by the second transformation of the pair. We used these counts to compute, for each transformation, how frequently each of the other transformation types followed it in a query.

We used these frequencies to create a state machine graph, as shown in Figure 3. Each node is a type of transformation, and each edge from transformation $A$ to a transformation $B$ indicates the number of times $B$ was used after $A$ as a fraction of the number of times $A$ was used. Also included as nodes are states representing the start of a query, before any command has been issued, and the end of a query, when no further commands are issued. The edges between these nodes can be thought of as transition probabilities that describe how likely a user is to issue transformation $B$ after having issued transformation $A$.

Using these graphs, we can discover typical log analysis pipelines employed by Splunk users. We exclude from presentation sequences with **Cache** transformations, as those have in most cases been automatically added to scheduled queries by Splunk to optimize them, as well as **Macro**s, because these can represent any transformation, so we do not learn much by including them. The remaining top transformation pipelines by weight (where the weight of a path is the product of its edges) are:

- **Filter**
- **Filter** | **Aggregate**
- **Filter** | **Filter** [3]
- **Filter** | **Augment** | **Aggregate**
- **Filter** | **Reorder**
- **Filter** | **Augment**

The preponderance of **Filter** transformations in typical pipelines is not surprising given that it is the most frequently applied transformation. It also makes sense in the context of log analysis – logging collects a great deal of information over the course of operation of a system, only a fraction of which is likely to be relevant to a given situation. Thus it is almost always necessary to get rid of this extraneous information. We investigate **Filter**, **Aggregate**, and **Augment** transformations in more detail in Section 6 to explain why these also appear in common pipelines.

These transformations sequences may seem simple compared to some log analysis techniques published in conferences like KDD or DSN [20, 25]. These pipelines more closely correspond to the simpler use cases described in the Dapper or Sawzall papers [34, 30]. There are many possible explanations for this: Most of the problems faced by log analysts may not be data mining or machine learning problems, and when they are, they may be difficult to map to published data mining and machine learning algorithms. Human intuition and domain expertise may be extremely competitive with state of the art machine learning and other techniques for a wide variety of problems – simple filters, aggregations and transformations coupled with visualizations are powerful tools in the hands of experts. Other reasons are suggested by user studies and first-hand industry experience [23, 38]. Users may prefer interpretable, easily adaptable approaches over black-boxes that require lots of mathematical expertise. It is worth further investigating the types of analysis techniques currently in widespread use and assess how the research on analysis techniques can better address practitioner needs.

We hypothesize that one important variable determining what transformation sequences are most often needed is the data type. Thus, we created more focused state machine graphs for two commonly analyzed source types by pulling out all queries that explicitly specified that source type[4]: Figure 4 shows the analysis applied to server access logs, used for web analytics (measuring traffic, referrals, and clicks). Figure 5 shows the results on operating system event logs (analyzing processes, memory and CPU usage). These figures suggest that indeed, query patterns can be expected to differ significantly depending on the type of data being analyzed. This could be due to the domain of the data, which could cause the types of questions asked to vary, or it could be due to the format of the data. For example web logs may have a more regular format, allowing users to avoid the convoluted processing required to normalize less structured data sources.

Other important factors likely include who the user is and what problems they are trying to solve. For example, in

[3]These can be thought of as one **Filter** that happened to be applied in separate consecutive stages.

[4]Source type can be specified in **Filter** transformations – this is what we looked for.

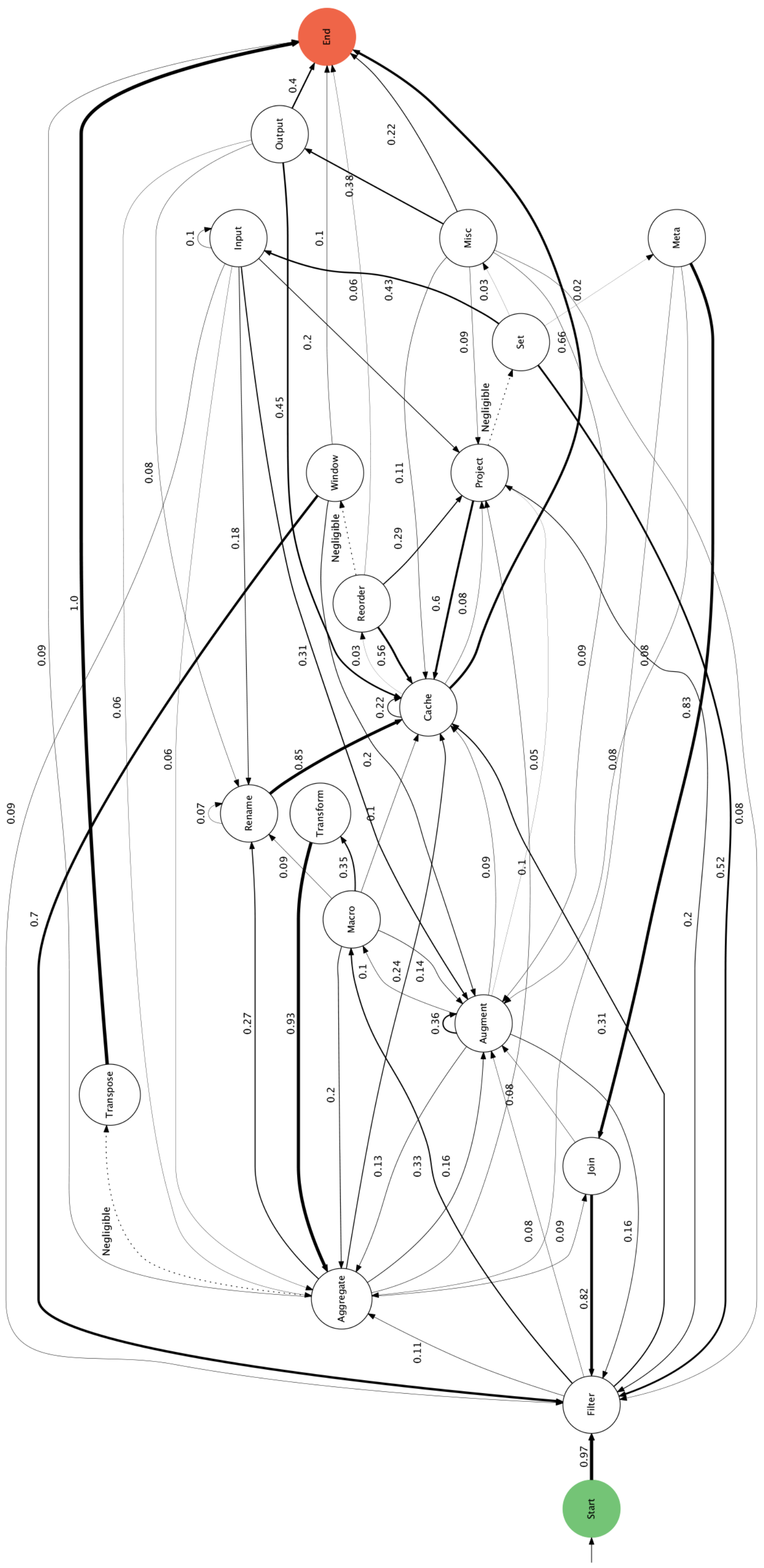


Figure 3: State machine diagram describing, for all distinct scheduled queries, the pairwise transition frequency between the command categories described in the text. Only edges with weight greater or equal to .05 are shown, for clarity.

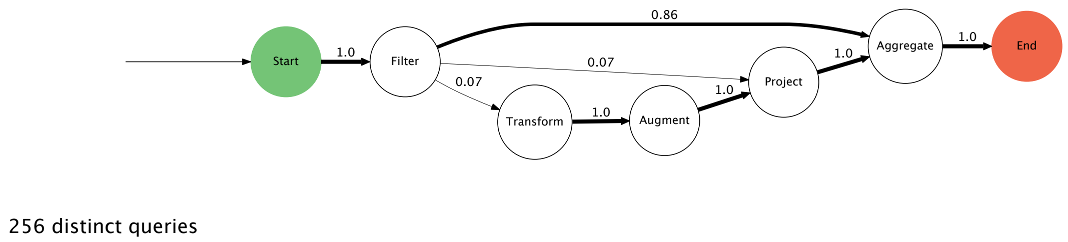


Figure 4: The pairwise transition frequency between transformations for web access log queries.

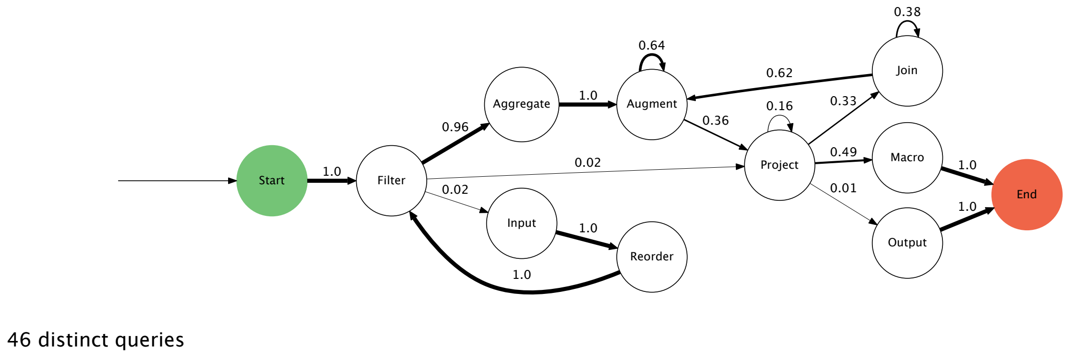


Figure 5: The pairwise transition frequency between transformations for OS event log queries.

the case of web access log data, an operations user will want to know, "Where are the 404s?[5] Are there any hosts that are down? Is there a spike in traffic that I should add capacity for?" A marketer will want to know, "What keywords are people searching today after my recent press release? What are the popular webinars viewed on my website?" A salesperson may ask, "Of the visitors today, how many are new versus returning, and how can I figure out whom to engage in a sales deal next based on what they're looking for on the web site?" Capturing this supplemental information from data analysis tools to include in the analysis would be useful for later tailoring tools to particular use cases. We have gathered some information about this (Section 7) but unfortunately we could not cross-reference this data with query data.

## 5.3 Longest Subsequences

To investigate what longer, possibly more complex, queries look like, we looked at the longest common subsequences of transformations (Table 4). Again, we excluded **Cache** and **Macro** transformations from presentation. We again see the preponderance of **Filter**, **Aggregate**, and **Augment** transformations. Beyond that, the most striking feature is the preponderance of **Augment** transformations, particularly in the longer subsequences. To gain more insight into exactly what such sequences of **Augment** transformations are doing, we look more closely at such transformations in the following section.

[5]404 is an HTTP standard response code indicating the requested resource was not found.

## 6 Cluster Analysis

Recall from Section 5 that three of the most common transformation types in log analysis are **Filter**, **Aggregate** and **Augment**. To find out more details about why and how such transformations are used, we clustered query stages containing these types of transformations, and then examined the distribution of transformations across these clusters. Clustering provides an alternative to manually looking through thousands of examples to find patterns. Similar conclusions would likely have been arrived at using manual coding techniques (i.e., content analysis), but this would have been more time-consuming.

In clustering these transformations, we investigate the following sets of questions:

- What are the different ways in which **Filter**, **Aggregate**, and **Augment** transformations are applied, and how are these different ways distributed?
- Can we identify higher-level tasks and activities by identifying related clusters of transformations? Do these clusters allow us to identify common workflow patterns? What can we infer about the user's information needs from these groups?
- How well do the *commands* in the Splunk query language map to the *tasks* users are trying to perform? What implications do the clusters we find have on data transformation language design?

| *Length* | *Count* | *% Queries* | *Subsequence* |
|---|---|---|---|
| 2 | 2866 | 16.77 | **Transform** \| **Aggregate** |
| 2 | 2675 | 6.13 | **Augment** \| **Augment** |
| 2 | 2446 | 14.06 | **Filter** \| **Aggregate** |
| 2 | 2170 | 12.70 | **Aggregate** \| **Rename** |
| 2 | 1724 | 8.42 | **Filter** \| **Augment** |
| 3 | 2134 | 12.49 | **Transform** \| **Aggregate** \| **Rename** |
| 3 | 1430 | 4.00 | **Augment** \| **Augment** \| **Augment** |
| 3 | 746 | 4.24 | **Aggregate** \| **Augment** \| **Filter** |
| 3 | 718 | 4.20 | **Aggregate** \| **Join** \| **Filter** |
| 3 | 717 | 4.20 | **Aggregate** \| **Project** \| **Filter** |
| 4 | 710 | 4.16 | **Aggregate** \| **Project** \| **Filter** \| **Rename** |
| 4 | 710 | 4.16 | **Transform** \| **Aggregate** \| **Augment** \| **Filter** |
| 4 | 694 | 2.71 | **Augment** \| **Augment** \| **Augment** \| **Augment** |
| 4 | 472 | 2.73 | **Filter** \| **Augment** \| **Augment** \| **Augment** |
| 4 | 234 | 1.37 | **Augment** \| **Augment** \| **Augment** \| **Project** |
| 5 | 280 | 1.62 | **Filter** \| **Augment** \| **Augment** \| **Augment** \| **Augment** |
| 5 | 222 | 1.30 | **Augment** \| **Augment** \| **Augment** \| **Augment** \| **Project** |
| 5 | 200 | 0.61 | **Augment** \| **Augment** \| **Augment** \| **Augment** \| **Augment** |
| 5 | 171 | 1.00 | **Augment** \| **Augment** \| **Augment** \| **Augment** \| **Filter** |
| 5 | 167 | 0.98 | **Filter** \| **Augment** \| **Augment** \| **Augment** \| **Aggregate** |
| 6 | 161 | 0.94 | **Augment** \| **Augment** \| **Augment** \| **Augment** \| **Filter** \| **Filter** |
| 6 | 160 | 0.94 | **Augment** \| **Augment** \| **Filter** \| **Filter** \| **Filter** \| **Augment** |
| 6 | 160 | 0.94 | **Augment** \| **Augment** \| **Augment** \| **Filter** \| **Filter** \| **Filter** |
| 6 | 148 | 0.87 | **Filter** \| **Augment** \| **Augment** \| **Augment** \| **Augment** \| **Filter** |
| 6 | 102 | 0.60 | **Augment** \| **Aggregate** \| **Augment** \| **Augment** \| **Augment** \| **Augment** |

Table 4: Longest common subsequences of transformations along with count of how many times such sequences appeared, and the percent of queries they appeared in.

To cluster each set of transformations, we:

(1) parsed each query (see: Section 4)
(2) extracted the stages consisting of the given transformation type,
(3) converted the stages into feature vectors,
(4) projected these feature vectors down to a lower dimensional space using PCA,
(5) projected these features further down into two dimensions, to allow visualization of the clusters, using t-SNE [37], and lastly
(6) manually identified and labeled clusters in the data.

Then, to count the number of transformations in each cluster, we use a random sample of 300 labeled examples from the clustering step to estimate the true proportion of stages in each cluster within 95% confidence intervals. [6]

## 6.1 Types of Filters

**Filter** stages primarily consist of the use of the `search` command, which almost all Splunk queries begin with, and which allows users to both select events from a source and filter them in a variety of ways. We clustered all distinct **Filter** stages and discovered 11 cluster types using 26 features[7] (Figure 6). Some of the clusters overlap, in that some examples could belong to more than one group. We discuss how we resolve this below.

The most common application of **Filter** is to use multi-predicate logical conditions to refine an event set, where these predicates are themselves filters of the other types, such as those that look for matches of a given field (e.g., `search status=404`), or those that look for any event containing a specified string (e.g., `search "Authentication failure for user: alspaugh"`). When a **Filter** could go into multiple categories, it was placed into this one, which also contains **Filter**s with many predicates of the same type in a statement with many disjunctions and negations. Thus, it is the largest category. Considering each filter predicate individually might be more informative; we leave that to future work.

Another common **Filter** pulls data from a given source, index, or host (like a `SELECT` clause in SQL). These resemble **Filter**s that look for a match on a given field, but return all events from a given source rather than all events with a specific value in a specific field.

Other types of filters include those that deduplicate events, and those that filter based on time range, index, regular expression match, or the result of a function eval-

[6] Assuming cluster distribution is multinomial with $k$ parameters $p_i$ we use the formula $n = \frac{k^{-1}(1-k^{-1})}{(.05/1.96)^2}$ (which assumes each cluster is equally likely) to estimate the sample size required to estimate the true parameters with a 95% confidence interval. The maximum required size was 246.

[7] See Section 11 for more information about the features used.

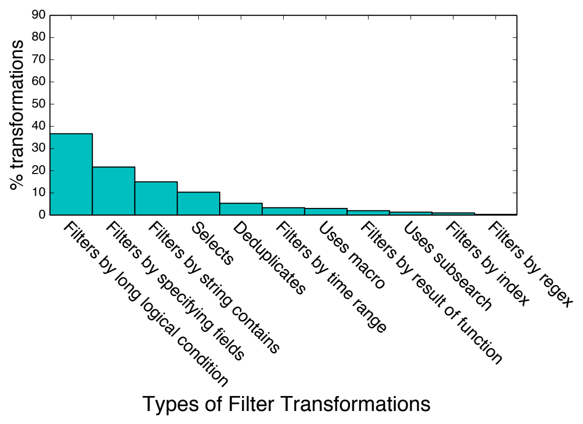


Figure 6: Distribution of different types of **Filter** transformations.

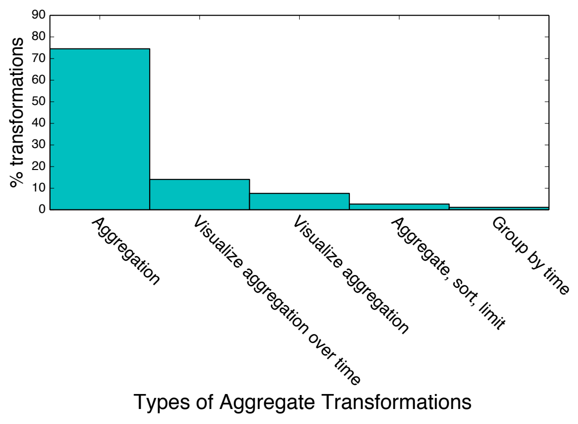


Figure 7: Distribution of different types of **Aggregate** transformations.

uation on the fields of each event. Lastly, some **Filter** transformations include the use of macros, others, the use of subsearches, the results of which are used as arguments to further constrain the current filter.

These use cases reveal several things:

- It is helpful to be able to simultaneously treat log data both as structured (field-value filters, similar to SQL `WHERE` clauses) and as unstructured (string-contains searches, similar to `grep`).
- Some commands in Splunk, like `search`, are heavily overloaded. A redesign of the language could make it easier to identify what users are doing, by bringing the task performed and the command invoked to perform it more in line with one another. For example, there could be a distinct command for each task identified above. This might also form a more intuitive basis on which to organize a data transformation language or interface, but would need to be evaluated for usability.
- Though it may appear that time range searches are not as prevalent as might have be suspected given the importance of the time dimension in log data, this is because the time range is most often encoded in other parameters that are passed along with the query. So time is still one of the most important filter dimensions for log analysis, but this is not reflected in these results.

## 6.2 Types of Aggregates

We discovered five **Aggregate** cluster types using 46 features (Figure 7). The most common **Aggregate** command is `stats`, which applies a specific aggregation function to any number of fields grouped by any number of other fields and returns the result. Most often, commonplace aggregation functions like `count`, `avg`, and `max` are used. Almost 75% of **Aggregate**s are of this type. Another 10% of **Aggregate**s do this, but then also prepare the output for visualization in a a chart rather than simply return the results (see the "Visualization" tab discussed in Section 3). Another common type of **Aggregate** is similar to these, but first buckets events temporally, aggregates each bucket, and displays the aggregated value over time in a histogram. Another type first aggregates, then sorts, then returns the top *N* results (e.g., `top user`). The last type groups by time, but not necessarily into uniformly sized buckets (e.g., when forming user sessions).

The takeaways from this are:

- Visualizing the results of aggregations is reasonably popular, though much of the time, simply viewing a table of the results suffices. Aggregations lead to the types of relational graphics that many people are familiar with, such as bar and line graphs [36]. Users might also appreciate having the ability to more easily visualize the result of **Filter** transformations as well; for example, using brushing and linking. [8]
- For log analysis, when visualization is used, it is more likely to visualize an aggregate value over buckets of time than aggregated over all time.

## 6.3 Types of Augments

**Augment**s add or transform a field for each event. The most commonly used such command is `eval`, which is another example of a heavily overloaded command. We discovered eight classes of **Augment** use by clustering over 127 features (Figure 8). These classes shed light onto the results of Section 5 and reveal what some of

[8]Brushing and linking is an interactive visualization technique wherein multiple views of data are linked and data highlighted in one view (i.e., a filter) appears also highlighted in the other view (i.e., a bar graph or heat map).

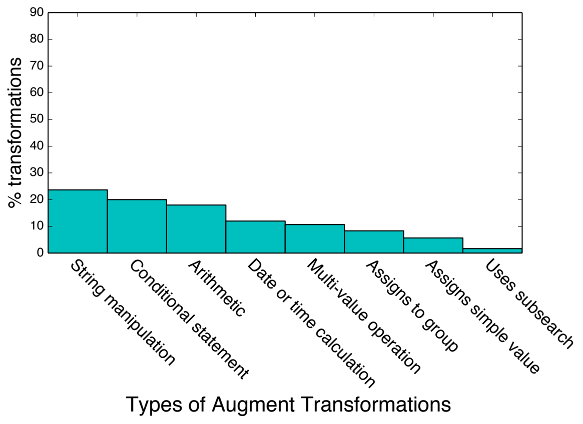


Figure 8: Distribution of different types of **Augment** transformations.

the long pipelines full of **Augment** transformations were likely doing.

The most common ways users transform their data are by manipulating strings (e.g., `eval name=concat(first, " ", last)`), conditionally updating fields (e.g., `eval type=if(status>=400, "failure", "success")`), performing arithmetic (e.g., `eval pct=cnt/total*100`), calculating date-time information (e.g., `eval ago=now()-_time`), applying multi-valued operations (e.g., `eval nitems=mvcount(items)`), or simple value assignments (e.g., `eval thresh=5`). Other `Augment` operations add a field that indicates which group an event belongs to and still others use the results of a subsearch to update a field.

These tasks reveal that:

- Aside from filtering and aggregation, much of log analysis consists of data munging (i.e., translating data from one format into another, such as converting units, and reformatting strings). This is supported by other studies of data analysis in general [28]. Such data munging transformations could be mined to create more intelligent logging infrastructure that outputs data in form already more palatable to end-users, or could be incorporated into an automated system that converts raw logs into nicely structured information. The more complicated transformations should be evaluated to identify whether the tool could be made more expressive.
- Just as with **Filter** transformations, here we observe heavily overloaded commands (i.e., `eval`). Refactoring functionality to clean up the mapping between tasks and commands would help here for the same reasons.

## 7 Usage Survey

The analytic results open many questions about usage goals that can best be answered by talking to the people who use the system. To this end, we administered a survey to Splunk sales engineers and obtained responses that describe the use cases, data sources, roles, and skill sets of 39 customer organizations. Note: these are not responses directly from customers, rather each sales engineer answered each question once for each of three customers, based on their firsthand knowledge and experience working with those customers. Figure 9 summarizes the results visually.

### 7.1 Survey Results

The main results are:

**User roles:** The bulk of Splunk users are in IT and engineering departments, but there is an important emerging class of users in management, marketing, sales, and finance. This may be because more business divisions are interleaving one or more machine generated log data sources for business insights.

**Programming experience:** Although most Splunk users are technically savvy, most only have limited to moderate amounts of programming experience.

**Splunk experience:** Surprisingly, many of the customers reported on did not consistently have expertise with Splunk, in fact, some users had no Splunk experience. This may be an artifact of the fact that the survey respondents were sales engineers, who may have opted to reply about more recent or growing customer deployments.

**Use cases:** Along with the main user roles, the main use cases are also IT-oriented, but, consistent with the other responses, Splunk is sometimes used to analyze business data.

**Data sources:** Correspondingly, the main type of data explored with Splunk is typical IT data: logs from web servers, firewalls, network devices, and so on. However, customers also used Splunk to explore sales, customer, and manufacturing data.

**Transformations applied:** Customers primarily use Splunk to extract strings from data, perform simple arithmetic, and manipulate date and time information. In some cases, customers perform more complex operations such as outlier removal and interpolation.

**Statistical sophistication:** Customers generally do not use Splunk to perform very complicated statistical analysis, limiting themselves to operations like computing descriptive statistics and looking for correlations. In one instance, a customer reported having a team of "math

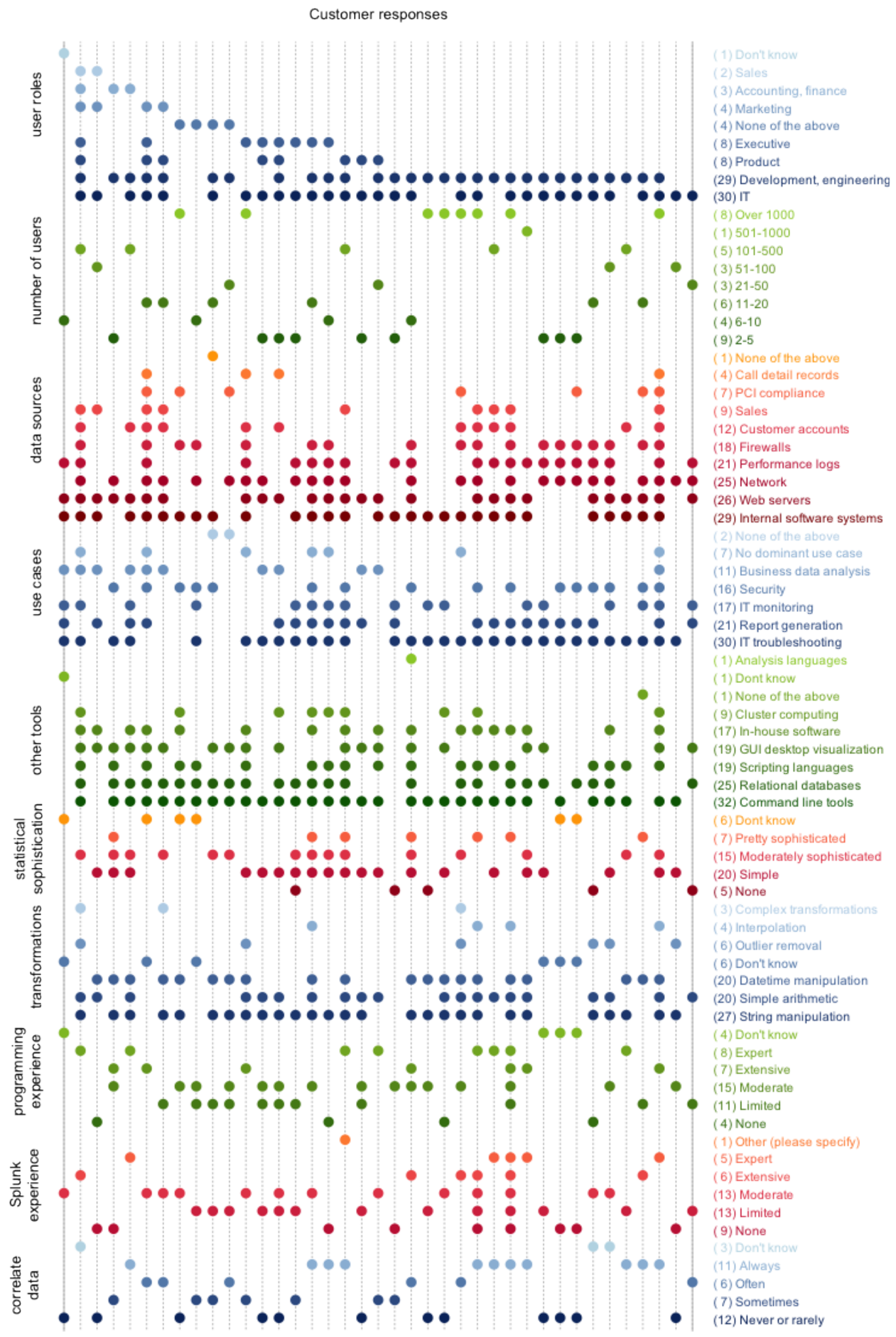


Figure 9: Summary of survey answers. Each vertical line represents a customer. Each colored grouping represents a different question and each row in the group represents one possible response to that question. A dot is present along a given column and row if the option corresponding to that row was selected for the question in that group, for the customer in that column.

junkies" that exported data out of Splunk, ran "very sophisticated batch analytics," and then imported those results back into Splunk for reporting.

**Data mash ups:** The degree to which customers combine data sources in their analysis varies across individual users and organizations. Some organizations almost always combine data sources for their analysis while a nearly equal number almost never do. This could be in part due to diversity in Splunk expertise and use cases.

**Other tools:** To better understand the ecosystem in which Splunk exists, we asked what other data analysis tools customers used. In keeping with their IT-oriented roles and use cases, command line tools are frequently used by most Splunk users, in addition to databases, scripting languages, and desktop visualization tools like Tableau. A significant number of customers used custom in-house applications for analyzing their data. A relatively small number used cluster computing frameworks or analysis languages like MATLAB.

Based on these results, we make the following predictions.

- IT and engineering professionals will be increasingly called upon to use their expertise working with machine data to aid other business divisions in their information-seeking needs, and will gain some expertise in these other domains as a result (deduced from user role and use case data).
- Classic tools of the trade for system administrators and engineers will be increasingly picked up by less technical users with other types of training, causing an evolution in both the features offered by the tools of the trade as well as the skills typically held by these other users (deduced from user role data). Although it is likely that more people in a growing variety of professions will learn how to program over the coming years, the market for log and data analysis tools that do not require programming experience will likely grow even faster (deduced from programming experience data).
- There is still no "one stop shop" for data analysis and exploration needs – customers rely on a variety of tools depending on their needs and individual expertise (based on the other tools data). This may be due to the existence of a well-established toolchain where different components are integrated into a holistic approach, not used disparately. Better understanding of which parts of different tools draw users would help both researchers and businesses that make data analysis products understand where to focus their energies.

## 8 Conclusion

In this paper we presented detailed, quantitative data describing the process of log analysis. While there have been a number of system administrator user studies, there have been few if any quantitative reports on traces of user behavior from an actual log analysis system at the level of detail we provide. In addition, we provide qualitative survey data for high-level context. Together these are important sources of information that can be used to to inform product design, guide user testing, construct statistical user models, and even create smart interfaces that make recommendations to users to enhance their analysis capabilities. We first summarize our main observations, then follow with a call to action for current tool builders and future researchers.

**Filtering:** In our observations, a large portion of log analysis activity in Splunk consists of filtering. One possible explanation is that log analysis is often used to solve problems that involve hunting down a few particular pieces of data – a handful of abnormal events or a particular record. This could include account troubleshooting, performance debugging, intrusion detection, and other security-related problems. Another possible explanation is that much of the information collected in logs, e.g., for debugging during development, is not useful for end-users of the system. In other words, logs include many different types of data logged for many different reasons, and the difference between signal and noise may depend on perspective.

**Reformatting:** Our analysis of **Augment** transformations suggested that most of these transformations were for the purpose of data munging, or reformatting and cleaning data. The prevalence of reformatting as a portion of log analysis activity is likely reflective of the fact that much log data is structured in an inconsistent, ad hoc manner. Taken together, the prevalence of filtering and reformatting activity in Splunk suggest that it may be useful for system developers to collaborate with the end users of such systems to ensure that data useful for the day-to-day management of such systems is collected. Alternatively, another possible explanation is that the Splunk interface is not as useful for other types of analysis. However, other reports indicate that indeed, much of data analysis in general does involve a lot of data munging [28].

**Summarization:** We observed that it is common in Splunk to **Aggregate** log data, which is a way of summarizing it. Summarization is a frequently-used technique in data analysis in general, and is used to create some of the more common graph types, such as bar charts and line graphs [36]. This suggests it may be useful to automatically create certain types of summarization to present to the user to save time. In log analysis with Splunk, summarizing with respect to the time dimension is an important use case.

**The complexity of log analysis activity:** We were not able to determine whether Splunk users make use of some of the more advanced data mining techniques proposed in the literature, such as techniques for event clustering and failure diagnosis [20, 25]. One possible explanation for this is that due to the complexity and variability of real world problems, as well as of logged information, designing one-size-fits-all tools for these situations is not feasible. Alternatively, such analyses may occur using custom code outside of Splunk or other analytics products as part of a large toolchain, into which we have little visibility. This idea is supported by some of the Splunk survey results (Section 7). Other possible explanations include lack of problems that require complicated solutions, lack of expertise, or requirements that solutions be transparent, which may not be the case for statistical techniques. It could also be the case that such techniques are used, but are drowned out by the volume of data munging activity. Finally, it may be that we simply were not able to find more complex analytics pipelines because programmatically identifying such higher-level activities from sequences of smaller, lower-level steps is a difficult problem.

**Log analysis outside of IT departments:** Our survey results also suggest that log analysis is not just for IT managers any longer; increasing numbers of non-technical users need to extract business insights from logs to drive their decision making.

## 9 Future Work

**Need for integrated provenance collection:** Understandably, most data that is logged is done so for the purpose of debugging systems, not building detailed models of user behavior [3]. This means that much of the contextual information that is highly relevant to understanding user behavior is not easily available, and even basic information must be inferred through elaborate or unreliable means [10]. We hope to draw attention to this issue to encourage solutions to this problem.

**Improving transformation representation:** In the process of analyzing the query data, we encountered difficulties relating to the fact that many commands in the Splunk language are heavily overloaded and can do many different things. For example, `stats` can both aggregate and rename data. When this is the case, we are more likely to have to rely on error-prone data mining techniques like clustering and classification to resolve ambiguities involved in automatically labeling user activities. If the mapping between analysis tasks and analysis representation (i.e., the analysis language) were less muddied, it would alleviate some of the difficulties of analyzing this activity data and pave the way for easier modeling of user behavior.

**Opportunities for predictive interfaces:** Thinking forward, detailed data on user behavior can be fed into advanced statistical models that return predictions about user actions. Studies such as the one we present are important for designing such models, including identifying what variables to model and the possible values they can take on. Other important variables to model could include who the user is, where their data came from, and what problems they are trying to solve. These could be used to provide suggestions to the user about what they might like to try, similar to how other recently successful tools operate [27].

**Further analyses of data analysis activity:** Finally, in this paper, we only presented data analysis activity from one system. It would be informative to compare this to data analysis activity from other systems, and on other types of data besides log data. Thus, we make our analysis code public so others can more easily adapt and apply our analysis to more data sets and compare results.

## 10 Acknowledgments

Thanks to David Carasso for being the go-to person for answering obscure questions about Splunk. Thanks to Ari Rabkin and Yanpei Chen for feedback on early drafts. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

## 11 Availability

The code used to generate the facts and figures presented in this paper, along with additional data that was not included due to lack of space, can be found at:

`https://github.com/salspaugh/lupe`

## References

[1] ALSPAUGH, S., ET AL. Towards a data analysis recommendation system. In *OSDI Workshop on Managing Systems Automatically*

*and Dynamically (MAD)* (2012).

[2] ALSPAUGH, S., ET AL. Building blocks for exploratory data analysis tools. In *KDD Workshop on Interactive Data Exploration and Analytics (IDEA)* (2013).

[3] ALSPAUGH, S., ET AL. Better logging to improve interactive data analysis tools. In *KDD Workshop on Interactive Data Exploration and Analytics (IDEA)* (2014).

[4] BARRETT, R., ET AL. Field studies of computer system administrators: Analysis of system management tools and practices. In *ACM Conference on Computer Supported Cooperative Work (CSCW)* (2004).

[5] BITINCKA, L., ET AL. Optimizing data analysis with a semi-structured time series database. In *OSDI Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)* (2010).

[6] CHEN, Y., ET AL. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[7] CHEN, Y., ET AL. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *International Conference on Very Large Databases (VLDB)* (2012).

[8] CHIARINI, M. Provenance for system troubleshooting. In *USENIX Conference on System Administration (LISA)* (2011).

[9] COUCH, A. L. ”standard deviations” of the ”average” system administrator. USENIX Conference on System Administration (LISA), 2008.

[10] GOTZ, D., ET AL. Characterizing users’ visual analytic activity for insight provenance. *IEEE Information Visualization Conference (InfoVis)* (2009).

[11] GRAY, J. Why do computers stop and what can be done about it? Tech. rep., 1985.

[12] HAUGERUD, H., AND STRAUMSNES, S. Simulation of user-driven computer behaviour. In *USENIX Conference on System Administration (LISA)* (2001).

[13] HREBEC, D. G., AND STIBER, M. A survey of system administrator mental models and situation awareness. In *ACM Conference on Computer Personnel Research (SIGCPR)* (2001).

[14] JANSEN, B. J., ET AL. Real life information retrieval: A study of user queries on the web. *SIGIR Forum* (1998).

[15] KLEIN, D. V. A forensic analysis of a distributed two-stage web-based spam attack. In *USENIX Conference on System Administration (LISA)* (2006).

[16] LAENDER, A. H. F., RIBEIRO-NETO, B. A., DA SILVA, A. S., AND TEIXEIRA, J. S. A brief survey of web data extraction tools. *SIGMOD Record* (2002).

[17] LAO, N., ET AL. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In *USENIX Conference on System Administration (LISA)* (2004).

[18] LOU, J.-G., ET AL. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Operating System Review* (2010).

[19] LOU, J.-G., FU, Q., WANG, Y., AND LI, J. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Operating System Review* (2010).

[20] MAKANJU, A. A., ET AL. Clustering event logs using iterative partitioning. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)* (2009).

[21] MARMORSTEIN, R., AND KEARNS, P. Firewall analysis with policy-based host classification. In *USENIX Conference on System Administration (LISA)* (2006).

[22] NORIAKI, K., ET AL. Semantic log analysis based on a user query behavior model. In *IEEE International Conference on Data Mining (ICDM)* (2003).

[23] OLINER, A., ET AL. Advances and challenges in log analysis. *ACM Queue* (2011).

[24] OLINER, A., AND STEARLEY, J. What supercomputers say: A study of five system logs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2007).

[25] OLINER, A. J., ET AL. Using correlated surprise to infer shared influence. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2010).

[26] OLSTON, C., ET AL. Pig latin: A not-so-foreign language for data processing. In *ACM International Conference on Management of Data (SIGMOD)* (2008).

[27] OTHERS, S. K. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Conference on Human Factors in Computing Systems (CHI)* (2011).

[28] OTHERS, S. K. Enterprise data analysis and visualization: An interview study. In *Visual Analytics Science & Technology (VAST)* (2012).

[29] PANG, R., ET AL. Characteristics of internet background radiation. In *ACM SIGCOMM Internet Measurement Conference (IMC)* (2004).

[30] PIKE, R., ET AL. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal* (2005).

[31] PINHEIRO, E., ET AL. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies (FAST)* (2007).

[32] POBLETE, B., AND BAEZA-YATES, R. Query-sets: Using implicit feedback and query patterns to organize web documents. In *International Conference on World Wide Web (WWW)* (2008).

[33] RICHARDSON, M. Learning about the world through long-term query logs. *ACM Transactions on the Web (TWEB)* (2008).

[34] SIGELMAN, B. H., AND OTHERS. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[35] SILVESTRI, F. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval* (2010).

[36] TUFTE, E., AND HOWARD, G. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

[37] VAN DER MAATEN, L., AND HINTON, G. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research (JMLR)* (2008).

[38] VELASQUEZ, N. F., ET AL. Designing tools for system administrators: An empirical test of the integrated user satisfaction model. In *USENIX Conference on System Administration (LISA)* (2008).

[39] XU, W., ET AL. Experience mining google’s production console logs. In *OSDI Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)* (2010).

# Realtime High-Speed Network Traffic Monitoring Using ntopng

Luca Deri, *IIT/CNR, ntop*    Maurizio Martinelli, *IIT/CNR*
Alfredo Cardigliano, *ntop*

## Abstract

Monitoring network traffic has become increasingly challenging in terms of number of hosts, protocol proliferation and probe placement topologies. Virtualised environments and cloud services shifted the focus from dedicated hardware monitoring devices to virtual machine based, software traffic monitoring applications.
This paper covers the design and implementation of ntopng, an open-source traffic monitoring application designed for high-speed networks. ntopng's key features are large networks real-time analytics and the ability to characterise application protocols and user traffic behaviour. ntopng was extensively validated in various monitoring environments ranging from small networks to .it ccTLD traffic analysis.

## 1. Introduction

Network traffic monitoring standards such as sFlow [1] and NetFlow/IPFIX [2, 3] have been conceived at the beginning of the last decade. Both protocols have been designed for being embedded into physical network devices such as routers and switches where the network traffic is flowing. In order to keep up with the increasing network speeds, sFlow natively implements packet sampling in order to reduce the load on the monitoring probe. While both flow and packet sampling is supported in NetFlow/IPFIX, network administrators try to avoid these mechanisms in order to have accurate traffic measurement. Many routers have not upgraded their monitoring capabilities to support the increasing numbers of 1/10G ports. Unless special probes are used, traffic analysis based on partial data results in inaccurate measurements.

Physical devices cannot monitor virtualised environments because inter-VM traffic is not visible to the physical network interface. Over the years however, virtualisation software developers have created virtual network switches with the ability to mirror network traffic from virtual environments into physical Ethernet ports where monitoring probes can be attached. Recently, virtual switches such as VMware vSphere Distributed Switch or Open vSwitch natively support NetFlow/sFlow for inter-VM communications [4], thus facilitating the monitoring of virtual environments. These are only partial solutions because either v5 NetFlow (or v9 with basic information elements only) or inaccurate, sample-based sFlow are supported. Network managers need traffic monitoring tools that are able to spot bottlenecks and security issues while providing accurate information for troubleshooting the cause. This means that while NetFlow/sFlow can prove a quantitative analysis in terms of traffic volume and TCP/UDP ports being used, they are unable to report the cause of the problems. For instance, NetFlow/IPFIX can be used to monitor the bandwidth used by the HTTP protocol but embedded NetFlow probes are unable to report that specific URLs are affected by large service time.

Today a single application may be based on complex cloud-based services comprised of several processes distributed across a LAN. Until a few years ago web applications were constructed using a combination of web servers, Java-based business logic and a database servers. The adoption of cache servers (e.g. memcache and redis) and mapReduce-based databases [5] (e.g. Apache Cassandra and MongoDB) increased the applications' architectural complexity. The distributed nature of this environment needs application level information to support effective network monitoring. For example, it is not sufficient to report which specific TCP connection has been affected by a long service time without reporting the nature of the transaction (e.g. the URL for HTTP, or the SQL query for a database server) that caused the bottleneck. Because modern services use dynamic TCP/UDP ports the network administrator needs to know what ports map to what application. The result is that traditional device-based traffic monitoring devices need to move towards software-based monitoring probes that increase network visibility at the user and application level. As this activity cannot be performed at network level (i.e. by observing traffic at a monitoring point that sees all traffic), software probes are installed on the physical/virtual servers where services are provided. This enables probes to observe the system internals and collect information (e.g. what user/process is responsible for a specific network connection) that would be otherwise difficult to analyse outside the system's context just by looking at packets.

Network administrators can then view virtual and cloud environments in real-time. The flow-based monitoring paradigm is by nature unable to produce real-time information [17]. Flows statistics such as throughput can be computed in flow collectors only for the duration of the flow, which is usually between 30 and 120 seconds (if not more). This means that using the flow paradigm, network administrators cannot have a real-time traffic view due to the latency intrinsic to this monitoring architecture (i.e. flows are first stored into the flow cache, then in the export cache, and finally sent to the collector) and also because flows can only report average values (i.e. the flow throughout can be computed by dividing the flow data volume for its duration) thus hiding, for instance, traffic spikes.

The creation of ntopng, an open-source web-based monitoring console, was motivated by the challenges of monitoring modern network topologies and the limitations of current traffic monitoring protocols. The main goal of ntopng is the ability to provide a real-time view of network traffic flowing in large networks (i.e. a few hundred thousand hosts exchanging traffic on a multi-Gbit link) while providing dynamic analytics able to show key performance indicators and bottleneck root cause analysis. The rest of the paper is structured as follow. Section 2 describes the ntopng design goals. Section 3 covers the ntopng architecture and its major software components. Section 4 evaluates the ntopng implementation using both real and synthetic traffic. Section 5 covers the open issues and future work items. Section 6 lists applications similar to ntopng, and finally section 7 concludes the paper.

## 2. ntopng Design Goals

*ntopng's* design is based on the experience gained from creating its predecessor, named ntop (and thus the name ntop next generation or ntopng) and first introduced in 1998. When the original ntop was designed, networks were significantly different. ntopng's design reflects new realities:

- Today's protocols are all IP-based, whereas 15 years ago many others existed (e.g. NetBIOS, AppleTalk, and IPX). Whereas only limited non-IP protocol support is needed, v4/v6 needs additional, and more accurate, metrics including packet loss, retransmissions, and network latency.
- In the past decade the number of computers connected to the Internet has risen significantly. Modern monitoring probes need to support hundreds of thousand of active hosts.
- While computer processing power increased in the last decade according to the Moore's law, system architecture support for increasing network interface speeds (10/10 Mbps to 10/40 today) has not always been proportional. As it will be later explained it is necessary to keep up with current network speeds without dropping packets.
- While non-IP protocols basically disappeared, application protocols have significantly increased and they still change rapidly as new popular applications appear (e.g. Skype). The association UDP/TCP port with an application protocol is no longer static, so unless other techniques, such as DPI (Deep Packet Inspection) [6] are in place, identifying applications based on ports is not reliable.
- As TLS (Transport Layer Security) [7] is becoming pervasive and no longer limited to secure HTTP, network administrators need partial visibility of encrypted communications.
- The HTTP protocol has greatly changed, as it is no longer used to carry, as originally designed, hypertext only. Instead, it is now used for many other purposes including audio/video streaming, firewall trespassing and in many peer-to-peer protocols. This means that today HTTP no longer identifies only web-related activities, and thus monitoring systems need to characterise HTTP traffic in detail.

In addition to the above requirements, ntopng has been designed to satisfy the following goals:

- Created as open-source software in order to let users study, improve, and modify it. Code availability is not a minor feature in networking as it enables users to compile and run the code on heterogeneous platforms and network environments. Furthermore, the adoption of this license allows existing open-source libraries and frameworks to be used by ntopng instead of coding everything from scratch as it often happens with closed-source applications.
- Operate at 10 Gbit without packet loss on a network backbone where user traffic is flowing (i.e. average packet size is 512 bytes or more), and support at least 3 Mpps (Million Packets/sec) per core on a commodity system, so that a low-end quad-core server may monitor a 10 Gbit link with minimal size packets (64 bytes).
- All monitoring data must be immediately available, with traffic counters updated in real-time without measurement latency and average counters that are otherwise typical of probe/collector architectures.
- Traffic monitoring must be fully implemented in software with no specific hardware acceleration re-

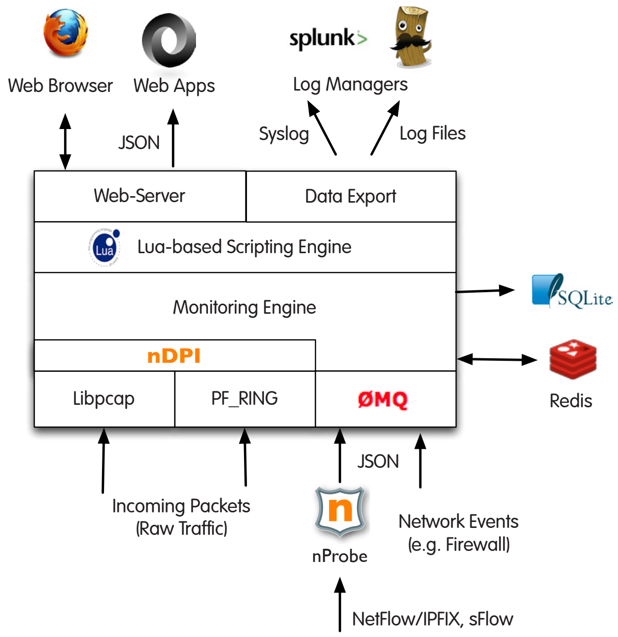
quirements. While many applications are now exploiting GPUs [8] or accelerated network adapters [9], monitoring virtual and cloud environments requires pure software-based applications that have no dependency on specific hardware and that can be migrated, as needed, across VMs.

- In addition to raw packet capture, ntopng must support the collection of sFlow/NetFlow/IPFIX flows, so that legacy monitoring protocols can also be supported.
- Ability to detect and characterise the most popular network protocols including (but not limited to) Skype, BitTorrent, multimedia (VoIP and streaming), social (FaceBook, Twitter), and business (Citrix, Webex). As it will be explained below, this goal has been achieved by developing a specific framework instead of including this logic within ntopng. This avoids the need of modifying ntopng when new protocols are added to the framework.
- Embedded web-based GUI based on HTML5 and dynamic web pages so that real-time monitoring data can be displayed using a modern, vector-based graphical user interface. These requirements are the foundation for the creation of rich traffic analytics.
- Scriptable and multi-threaded monitor engine so that dynamic web pages can be created and accessed by multiple clients simultaneously.
- Efficient monitoring engine not only in terms of packet processing capacity, but in its ability to operate on a wide range of computers, including low-power embedded systems as well as multi-core high-end servers. Support of low-end systems is necessary in order to embed ntopng into existing network devices such as Linux-based routers. This feature is to provide a low-cost solution for monitoring distributed and SOHO (Small Office Home Office) networks.
- Ability to generate alerts based on traffic conditions. In particular the alert definition should be configurable my means of a script, so that users can define their own conditions for triggering alerts.
- Integration with the system where traffic is observed, so that on selected servers, it is possible to correlate network events with system processes.

The following section covers in detail the ntopng software architecture and describes the various components on which the application is layered.

## 3. ntopng Software Architecture

ntopng is coded in C++ which enables source code portability across systems (e.g. X86, MIPS and ARM) and clean class-based design, while granting high execution speed.



1. ntopng Architecture

ntopng is divided in four software layers:

- Ingress data layer: monitoring data can be raw packets captured from one or more network interfaces, or collected NetFlow/IPFIX/sFlow flows after having been preprocessed.
- Monitoring engine: the ntopng core responsible for processing ingress data and consolidating traffic counters into memory.
- Scripting engine: a thin C++ software layer that exports monitoring data to Lua-based scripts.
- Egress data layer: interface towards external application that can access real-time monitoring data.

### 3.1. Ingress Data Layer

The ingress layer is responsible for receiving monitoring data. Currently three network interfaces are implemented:

- libpcap Interface: capture raw packets by means of the popular libpcap library.
- PF_RING Interface: capture raw packets using the open-source PF_RING framework for Linux systems [10] developed by ntop for enhancing both packet capture and transmission speed. PF_RING is divided in two parts: a kernel module that efficiently interacts with the operating system and network drivers, and a user-space library that interacts with the kernel mod-

ule, and implements an API used by PF_RING-based applications. The main difference between libpcap and PF_RING, is that when using the latter it is possible to capture/receive minimum size packets at 10 Gbit with little CPU usage using commodity network adapters. PF_RING features these performance figures both on physical hosts and on Linux KVM-based virtual machines, thus paving the way to line-rate VM-based traffic monitoring.

- ØMQ Interface. The ØMQ library [12] is an open-source portable messaging library coded in C++ that can be used to implement efficient distributed applications. Each application is independent, runs on its own memory space, and it can be deployed on the same host where ntopng is running, or on remote hosts. In ntopng it has been used to receive traffic-related data from distributed systems. ntopng creates a ØMQ socket and waits for events formatted as JSON (JavaScript Object Notation) [16] strings encoded as "<element id>": "<value>", where <element id> is a numeric identifier as defined in the NetFlow/IPFIX RFCs. The advantages of this approach with respect of integrating a native flow collector, are manyfold :
  - The complexities of flow standards are not propagated to ntopng, because open-source applications such as nProbe [13] act as a proxy by converting flows into JSON strings delivered to ntopng via ØMQ.
  - Any non-flow network event can be collected using this mechanism. For instance, Linux firewall logs generated by netfilter, can be parsed and sent to ntopng just like in commercial products such as Cisco ASA.

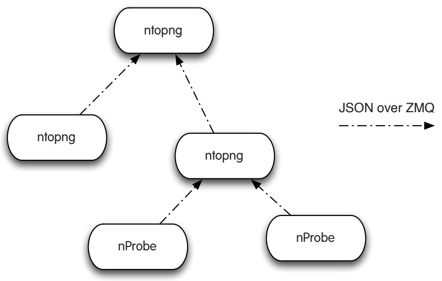
Contrary to what happens with flow-based tools where the probe delivers flows to the collector, when used over ØMQ ntopng acts as a consumer. As depicted in Fig 1., ntopng (as flow consumer) connects to nProbe (that acts as flow producer) that acts as flow probe or proxy (i.e. nProbe collects flows sent by a probe and forwards them to ntopng). Flows are converted into JSON messages that are read by ntopng via ØMQ.

```
{"IPV4_SRC_ADDR":"10.10.20.15","IPV4_D-
ST_ADDR":"192.168.0.200","IPV4_NEXT_HOP":
"0.0.0.0","INPUT_SNMP":0,"OUTPUT_SNMP":
0,"IN_PKTS":12,"IN_BYTES":
11693,"FIRST_SWITCHED":
1397725262,"LAST_SWITCHED":
1397725262,"L4_SRC_PORT":
80,"L4_DST_PORT":50142,"TCP_FLAGS":
27,"PROTOCOL":6,"SRC_TOS":0,"SRC_AS":
3561,"DST_AS":0,"TOTAL_FLOWS_EXP":8}
```

2. NetFlow/IPFIX flow converted in JSON by nProbe

The JSON message uses as field key the string values defined in the NetFlow RFC [2], so in essence this is a one-to-one format translation from NetFlow to JSON. The combination of ØMQ with redis can also be used to employ ntopng as a visualisation console for non-packet related events. For instance at the .it ccTLD, ntopng receives JSON messages via ØMQ from domain name registration system that are accessed via the Whois [35], DAS (Domain Availability Service) [36] and EPP (Extensible Provisioning Protocol) [37] protocols. Such protocol messages are formatted in JSON using the standard field key names defined in the NetFlow RFC, and add extra fields for specifying custom information not defined in the RFC (e.g. the DNS domain name under registration). In essence the idea is that ntopng can be used to visualise any type of network related information, by feeding into it (via ZMQ) data formatted in JSON. In case the JSON stream carries unknown fields, ntopng will just be able to display the field on the web interface but the data processing will not be affected (i.e. messages with unknown field names will not be discarded).

The use of JSON not only allows application complexity to be reduced but it also promotes the creation of arbitrary application hierarchies. In fact each ntopng instance can act both as a data consumer or producer.



3. Cluster of ntopng and nProbe applications.

When a flow is expired, ntopng propagates the JSON-formatted flow information to the configured instance up one hierarchy. Each ntopng instance can collect traffic information from multiple producers, and each producer can send traffic information to multiple consumers. In essence using this technique it is possible to create a (semi-) centralised view of a distributed monitoring environment simply using ntopng without any third party tool or process that might make the overall architecture more complex.

The overhead introduced by JSON is minor, as ntopng can collect more than 20k flows/sec per interface. In case more flows need to be collected, ntopng can be configured to collect flows over multiple interfaces. Each ingress interface is self-contained with no cross-dependencies. When an interface is configured at startup, ntopng creates a data polling thread bound to it. All the data structures, used to store monitoring data are defined per-interface and are not global to ntopng. This has the advantage that each network interface can operate independently, likely on a different CPU core, to create a scalable system. This design choice is one of the reasons for ntopng's superior data processing performance as will be shown in the following section.

### 3.2. Monitoring Engine

Data is consolidated in ntopng's monitoring engine. This component is implemented as a single C++ class that is instantiated, one per ingress interface, in order to avoid performance bottlenecks due to locking when multiple interfaces are in use. Monitoring data is organised in flows and hosts, where by flow we mean a set of packets having the same 6-tuple (VLAN, Protocol, IP/Port Source/Destination) and not as defined in flow-based monitoring paradigms where flows have additional properties (e.g. flow duration and export). In ntopng a flow starts when the first packet of the flow arrives, and it ends when no new data belonging to the flow is observed for some time. Regardless of the ingress interface type, monitoring data is classified in flows. Each ntopng flow instance references two host instances (one for flow source and the other for flow destination) that are used to keep statistics about the two peers. This is the flow lifecycle:

- When a packet belonging to a new flow is received, the monitoring engine decodes the packet and searches a flow instance matching the packet. If not found, a flow instance is created along with the two flow host instances if not existing.
- The flow and host counters (e.g. bytes and packets) are updated according to the received packets.
- Periodically ntopng purges flows that have been idle for a while (e.g. 2 minutes with no new traffic received). Hosts with no active flows that have also been idle for some time are also purged from memory.

Purging data from memory is necessary to avoid exhausting all available resources and discard information no longer relevant. However this does not mean that host information is lost after data purge but that it has been moved to a secondary cache. Fig. 1 shows that monitoring engine connects with Redis [14], a key-value in-memory data store. ntopng uses redis as data cache where it stores:

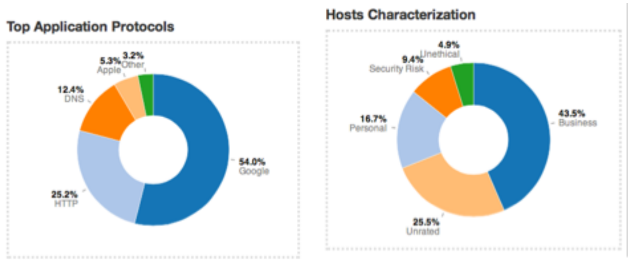
- JSON-serialised representation of hosts that have been recently purged from memory, along with their traffic counters. This allows hosts to be restored in memory whenever they receive fresh traffic while saving ntopng memory.
- In case ntopng has been configured to resolve IP address into symbolic names, redis stores the association numeric-to-symbolic address.
- ntopng configuration information.
- Pending activities, such as the queue of numeric IPs, waiting to be resolved by ntopng.

Redis has been selected over other popular databases (e.g. MySQL and memcached) for various reasons:

- It is possible to specify whether stored data is persistent or temporary. For instance, numeric-to-symbolic data is set to be volatile so that it is automatically purged from redis memory after the specified duration with no action from ntopng. Other information such as configuration data is saved persistently as it happens with most databases.
- Redis instances can be federated. As described in [15] ntopng and nProbe instances can collaborate and create a microcloud based on redis. This microcloud consolidates the monitoring information reported by instances of ntopng/nProbe in order to share traffic information, and effectively monitor distributed networks.
- ntopng can exploit the publish/subscribe mechanisms offered by redis in order to be notified when a specific event happens (e.g. a host is added to the cache) and thus easily create applications that execute specific actions based on triggers. This mechanism is exploited by ntopng to distribute traffic alerts to multiple consumers using the microcloud architecture described later on this section.

In ntopng all the objects can be serialised in JSON. This design choice allows them to be easily stored/retrieved from redis, exported to third party applications (e.g. web apps), dumped on log files, and immediately used in web pages though Javascript. Through JSON object serialisation it is possible to migrate/replicate host/flow objects across ntopng instances. As mentioned above, JSON serialisation is also used to collect flows from nProbe via ØMQ and import network traffic information from other sources of data.

In addition to the 6-tuple, ntopng attempts to detect the real application protocol carried by the flow. For collected flows, unless specified into the flow itself, the application protocol is inferred by inspecting the IP/ ports used by the flows. For instance, if there is a flow from a local PC to a host belonging to the Dropbox Inc network on a non-known port, we assume that the flow uses the dropbox protocol. When network interfaces operate on raw packets, we need to inspect the packets' payload. ntopng does application protocol discovery using nDPI [18], a home-grown GPLv3 C library for deep packet inspection. To date nDPI recognises over 170 protocols including popular ones such as BitTorrent, Skype, FaceBook, Twitter[1], Citrix and Webex. nDPI is based on an a protocol-independent engine that implements services common to all protocols, and protocol-specific dissectors that analyse all the supported protocols. If nDPI is unable to identify a protocol based on the packet payload it can try to infer the protocol based on the IP/port used (e.g. TCP on port 80 is likely to be HTTP). nDPI can handle both plain and encrypted traffic: in the case of SSL (Secure Socket Layers) nDPI decodes the SSL certificate and it attempts to match the server certificate name with a service. For instance encrypted traffic with server certificate *.amazon.com is traffic for the popular Amazon web site, and *.viber.-com identifies the traffic produced by the mobile Viber application. The library is designed to be used both in user-space inside applications like ntopng and nProbe, and in the kernel inside the Linux firewall. The advantage of having a clean separation between nDPI and ntopng is that it is possible to extend/modify these two components independently without polluting ntopng with protocol-related code. As described in [19], nDPI accuracy and speed is comparable to similar commercial products and often better than other open-source DPI toolkits.



4. Application Protocol Classification vs. Traffic Characterisation

In addition to DPI, ntopng is able to characterise traffic based on its nature. An application's protocol describes how data is transported on the wire, but it tells nothing about the nature of the traffic. To that end ntopng natively integrates Internet domain categorisation services freely provided to ntopng users by http://block.si. For instance, traffic for cnn.com is tagged as "News and Media", whereas traffic for FaceBook is tagged as "Social". It is thus possible to characterise host behaviour with respect to traffic type, and thus tag hosts that perform potentially dangerous traffic (e.g. access to sites whose content is controversial or potentially insecure) that is more likely to create security issues. This information may also be used to create host traffic patterns that can be used to detect potential issues, such as when a host changes its traffic pattern profile over time; this might indicate the presence of viruses or unwanted applications. Domain categorisation services are provided as a cloud-service and accessed by ntopng via HTTP. In order to reduce the number of requests and thus minimise the network traffic necessary for this service, categorisation responses are cached in redis similar to the IP/host DNS mapping explained earlier in this section.

In addition to domain classification, ntopng can also identify hosts that are previously marked as malicious. When specified at startup, ntopng can query public services in order to track harvesters, content spammers, and other suspicious activities. As soon as ntopng detects traffic for a new host not yet observed, it issues a DNS query to the Project Honeypot [34] that can report information about such host. Similar to what happens with domain categorisation, ntopng uses redis to cache responses (the default cache duration is 12 hours) in order to reduce the number of DNS queries. In case a host has been detected as malicious, ntopng triggers an alert and reports in the web interface the response returned that includes threat score and type.

### 3.3. Scripting Engine

The scripting engine sits on top of the monitoring engine, and it implements a Lua-based API for scripts that need to access monitoring data. ntopng embeds the Lua JIT (Just In Time) interpreter, and implements two Lua classes able to access ntopng internals.

- interface: access to interface-related data, and to flow and host traffic statistics.
- ntop: it allows scripts to interact with ntopng configuration and the redis cache.

The scripting engine decouples data access from traffic processing through a simple Lua API. Scripts are exe-

[1] Please note that technically FaceBook is HTTP(S) traffic from/to FaceBook Inc. servers. This also applies to Twitter traffic. However nDPI assigns them a specific application protocol Id in order to distinguish them from plain HTTP(S) traffic.

cuted when they are requested though the embedded web server, or based on periodic events. ntopng implements a small cron daemon that runs scripts periodically with one second granularity. Such scripts are used to perform periodic activities (e.g. dump the top hosts that sent/received traffic in the last minute) as well data housekeeping. For instance every night at midnight, ntopng runs a script that dumps on a SQLite database all the hosts monitored during the last 24 hours; this way ntopng implements a persistent historical view of the recent traffic activities.

The clear separation of traffic processing from application logic has been a deliberate choice in ntopng. The processing engine (coded in C++) has been designed to do simple traffic-related tasks that have to be performed quickly (e.g. receive a packet, parse it, update traffic statistics and move to the next packet). The application logic instead can change according to user needs and preferences and thus it has been coded with scripts that access the ntopng core by means of the Lua API. Given that the Lua JIT is very efficient in terms of processing speed, this solution allows users to modify the ntopng business logic by simply changing scripts instead of modifying the C++ engine.

```
dirs = ntop.getDirs()

package.path = dirs.installdir .. "/
scripts/lua/modules/?.lua;" .. package.-
path

require "lua_utils"

sendHTTPHeader('text/html')

print('<html><head><title>ntop</title></
head><body>Hello ' .. os.date("%d.%m.
%Y"))

print('<li>Default ifname = ' .. inter-
face.getDefaultIfName()
```

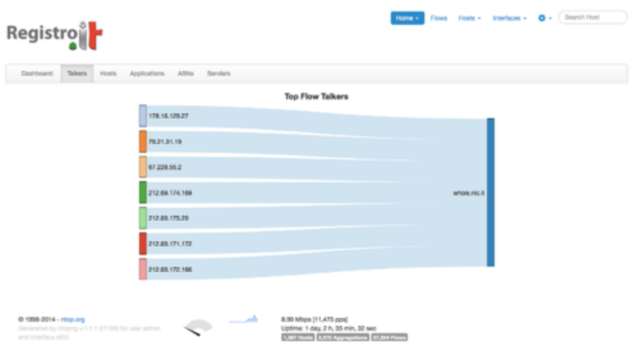
5. Simple ntopng Lua Script

When a script accesses an ntopng object, the result is returned to the Lua script as a Lua table object. In no case Lua references C++ object instances directly, thus avoiding costly/error-prone object locks across languages. All ntopng data structures are lockless, and Lua scripts lock C++ data structures only if they scan the hosts or flows hash. Multiple scripts can be executed simultaneously, as the embedded Lua engine is multithreaded and reentrant.

It is worth to remark that the scripting engine is used only to report information produced by the monitoring engine and for other periodic activities such alert triggering, and not to process monitoring information. The design choice of having a C++-based monitoring engine with Lua scripts for reporting data, is a good compromise in terms of performance and flexibility. This because it allows to preserve the engine efficiency while enabling users to customise the GUI without having to modify the monitoring engine.

### 3.4. Egress Data Layer

ntopng exports monitoring data through the embedded HTTP server that can trigger the execution of Lua scripts. The web GUI is based on the Twitter Bootstrap JavaScript framework [20] that enables the creation of dynamic web pages with limited coding. All charts are based on the D3.JS [25] that features a rich set of HTML5 components that can be used to represent monitoring data in an effective way.



6. ntopng HTML5 Web Interface

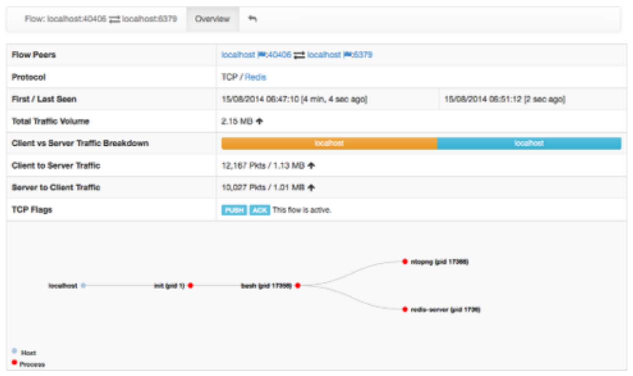
The embedded web server serves static pages containing JavaScript code that triggers the execution of Lua scripts. Such scripts access ntopng monitoring data and return their results to the web browser in JSON format. Web pages are dynamically updated every second by the JavaScript code present in the web pages, that requests the execution of Lua scripts.

As stated earlier in this section, ntopng can manipulate JSON objects natively, thus enabling non-HTML applications to use ntopng as a server for network traffic data as well. Through Lua scripts, it is possible to create REST-compliant (Representational State Transfer) [21] Web services on top of ntopng.

Another way to export monitoring data from ntopng, is by means of log files. With the advent of high-capacity log processing applications such as Splunk and ElasticSearch/Logstash, ntopng can complement traditional service application logs with traffic logs. This allows network administrators to correlate network traffic information to service status. Export in log files is performed through Lua scripts that can access the monitoring engine and dump data into log files or send it via the syslog protocol [22], a standard for remote message logging.

### 3.5. System Integration

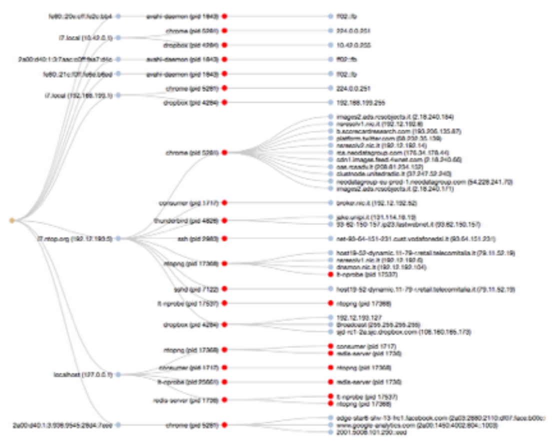
Monitoring network traffic is not enough to have a complete picture of the system. Often it is necessary to correlate traffic with system information. For this reason ntopng has been integrated (on Linux only for the time being) with a nProbe plugin named *sprobe*. Such plugin is based on sysdig [38], a Linux kernel module and library that allows system events to be captured from user-space, similar to what happens with libpcap. The plugin listens to system events such as creation/deletion of a network connection by intercepting system calls such as accept() and connect(). When such events happen, nProbe exports process information provides via ØMQ this event information formatted in JSON to ntopng or via NetFlow/IPFIX to standard flow collectors. This way ntopng, for those systems where nProbe is active, can associate a communication flow with a process name.



Flow: localhost:40406 ⇄ localhost:6379 | Overview

| | | |
|---|---|---|
| Flow Peers | localhost :40406 ⇄ localhost :6379 | |
| Protocol | TCP / Redis | |
| First / Last Seen | 15/08/2014 06:47:10 [4 min, 4 sec ago] | 15/08/2014 06:51:12 [2 sec ago] |
| Total Traffic Volume | 2.15 MB | |
| Client vs Server Traffic Breakdown | localhost | localhost |
| Client to Server Traffic | 12,167 Pkts / 1.13 MB | |
| Server to Client Traffic | 10,027 Pkts / 1.01 MB | |
| TCP Flags | PUSH ACK This flow is active. | |

7. ntopng Report on Process Information received from sProbe



8. ntopng Processes Interaction Report

In addition to network information, sProbe reports information about the process itself by looking at the /proc filesystem.

In particular it reports information about the memory being used (current and max memory), the number of VM page faults, and process CPU usage.

If multiple sProbe instances feed ntopng, it is possible to correlate the information across system. For instance it is possible to see that Google Chrome on host 192.168.2.13 is connected to ntopng running on system 10.10.12.18. As flow information is periodically exported by sProbe and not just at the beginning/end of the flow, ntopng can also report process activities over time thus combing network with system monitoring.

## 4. Evaluation

ntopng has been extensively tested by its users in various heterogeneous environments. This section reports the results of some validation tests performed on a lab using both synthetic and real traffic captured on a network.

| Hosts Number | PPS | Gbit | CPU Load | Packet Drops | Memory Usage |
|---|---|---|---|---|---|
| 350 | 1’735’000 | 10 | 80% | 0% | 27 MB |
| 600 | 1’760’000 | 10 | 80% | 0% | 29 MB |

9. Tests Using Real Traffic (Average Packet Size 700 bytes)

The tests have been performed using ntopng v.1.1.1 (r7100) on a system based on a low-end Intel Xeon E3-1230 running at 3.30 GHz. ntopng monitors a 10 Gbit Intel network interface using PF_RING DNA v.5.6.1. The traffic generator and replay is *pfsend*, an open-source tool part of the PF_RING toolset. In case of real traffic, pfsend has reproduced in loop at line rate the pcap file captured on a real network. In the case of synthetic traffic, pfsend has generated the specified number of packets by forging packets with the specified hosts number. Please note that increasing the number of active hosts also increases the number of active flows handled by ntopng.

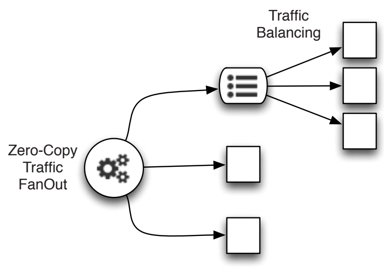
The previous table reports the test with traffic captured on a real network and reproduced by pfsend at line rate. The result shows that ntopng is able to monitor a fully loaded 10 Gbit link without loss and with limited memory usage. Considered that the test system is a low-end server, this is a great result, which demonstrates that it is possible to monitor a fully loaded link with real traffic using commodity hardware and efficient software. Using synthetic traffic we have studied how the number of monitored hosts affects the ntopng performance. Increasing the cardinality of hosts and flows, ntopng has to perform heavier operations during data structure lookup and periodic data housekeeping.

| Packet Size | 64 bytes | 128 bytes | 512 bytes | 1500 bytes |
|---|---|---|---|---|
| **Hosts Number** | **Processed PPS** | **Processed PPS** | **Processed PPS** | **Processed PPS** |
| **100** | 8'100'000 | 8'130'000 | 2'332'090 | 2'332'090 |
| **1'000** | 7'200'000 | 6'580'000 | 2'332'090 | 820'210 |
| **10'000** | 5'091'000 | 4'000'000 | 2'332'090 | 819'000 |
| **100'000** | 2'080'000 | 2'000'000 | 1'680'000 | 819'000 |
| **1'000'000** | 17'800 | 17'000 | 17'000 | 17'000 |

10. Synthetic Traffic: Packet Size/Hosts Number vs. Processed Packets (PPS)

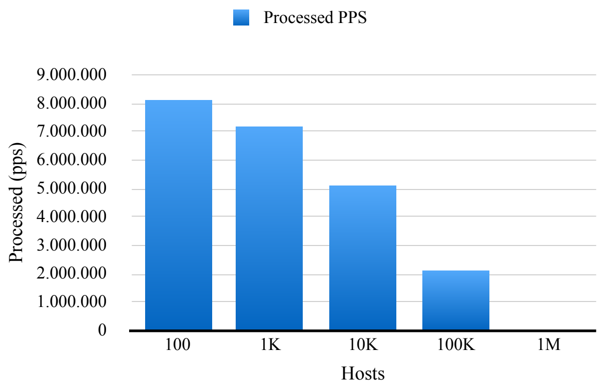
The above figure shows how the number of hosts and packet size influence the number of processes packets. Packet capture is not a bottleneck due to the use of PF_RING DNA. However, ntopng's processing engine performance is reduced in proportion with the number of active hosts and flows. Although networks usually have no more than a few thousand active hosts, we tested ntopng's performance across many conditions ranging from a small LAN (100 hosts), a medium ISP (10k hosts) and large ISP (100k hosts) to a backbone (1M hosts). The setup we used was worst case, because, in practice it is not a good choice to send traffic from a million hosts to the same ntopng monitoring interface.

The PF_RING library named *libzero* has the ability to dispatch packets in zero-copy to applications, threads and KVM-based VMs.



11. pfdnacluster_master: PF_RING Zero Copy Traffic Balancing

The open-source application *pfdnacluster_master*[2] can read packets from multiple devices and implement zero-copy traffic fan-out (i.e. the same ingress packet is replicated to various packet consumers) and/or traffic balancing. Balancing respects the flow-tuple, meaning that all packets of flow X will always be sent to the egress virtual interface Y; this mechanisms works also with encapsulated traffic such as GTP traffic used to encapsulate user traffic in mobile networks [23]. This application can create many egress virtual interfaces not limited by the number and type of physical interfaces from which packets are received.

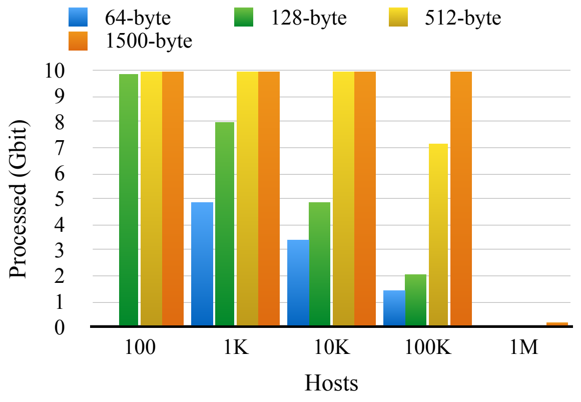


12. Synthetic Traffic: Hosts Number vs. Processed Packets (PPS)

Thanks to PF_RING it is possible to balance ingress traffic to many virtual egress interfaces, all monitored by the same ntopng process that binds each packet processing thread to a different CPU core. This practice enables concurrent traffic processing, as it also reduces the number of hosts/flows handled per interface, thus increasing the overall performance. In our tests we have decided to measure the maximum processing capability per interface so that we can estimate the maximum ntopng processing capability according to the number of cores available on the system. Using the results reported in the previous figures, using real traffic balanced across multiple virtual interfaces, ntopng could easily monitor multi-10 Gbit links, bringing real-time traffic monitoring to a new performance level.

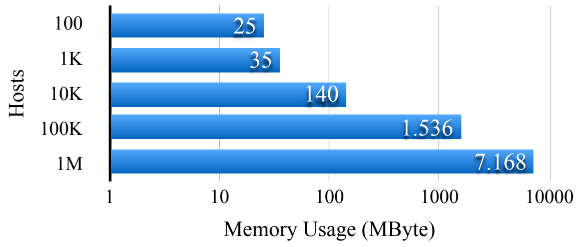
The previous chart above depicts the data in Fig. 10 by positioning the processing speed with respect to the number of hosts. As reported in Fig. 9 using real traffic on a full 10 Gbit link we have approximately 1.7 Mpps. At that ingress rate, ntopng can successfully handle more than 100K active hosts per interface, thus making it suitable for a large ISP. The following figure shows the same information as Fig. 12 in terms of Gbit instead of Pps (Packet/sec).

[2] Source code available at https://svn.ntop.org/svn/ntop/trunk/PF_RING/userland/examples_libzero/pfdnacluster_master.c



13. Synthetic Traffic: Hosts Number vs. Processed Packets (Gbit)

Similar to processing performance, ntopng's memory usage is greatly affected by the number of active hosts and flows. As the traffic is reproduced in loop, hosts and flows are never purged from memory as they receive continuously fresh new data.



14. Hosts Number vs. Memory Usage

Memory usage ranges from 20 MB for 100 active hosts, to about 7 GB for 1 million hosts. Considered that low-end ARM-based systems [26] such as the RaspberryPI and the BeagleBoard feature 512 MB of memory, their use enables the monitoring of ~40k simultaneous hosts and flows. This is an effective price-performance ratio given the cost ($25) and processing speed of such devices. ntopng code compiles out of the box on these devices and also on the low-cost (99$) Ubiquity EdgeMax router where it is able to process 1 Mpps. Both nProbe and ntopng run on the above mentioned platforms and there are commercial companies that deploy such small boxes in order to complement traditional ping/traceroute/iperf remote monitoring with real-time traffic visualisation as produced by ntopng.

## 5. Open Issues and Future Work

While we have successfully run ntopng on systems with limited computation power, we are aware that in order to monitor a highly distributed network such as cloud system, it is necessary to consolidate all data in a central location. As both VMs and small PCs have limited storage resources, we are working on the implementation of a cloud-based storage system that allows distributed ntopng instances to consolidate monitoring data onto the same data repository.

Another future work item is the ability to further characterise network traffic by assigning it a security score. Various companies provide something called IP reputation [24] a number which the danger potential of a given IP. We are planning to integrate cloud-based reputation services into ntopng similarly to what we have done for domain categorisation. This would enable spot monitoring of hosts that generate potentially dangerous network traffic.

Finally we are planning to introduce data encryption and authentication in ZMQ communications. This problem was not properly addressed in ZMQ until recent library versions, and thus it needs also to be integrated into ntopng in order to guarantee secure data sharing across ntopng applications.

## 6. Related Work

When the original ntop had been introduced in 1998 it was the first traffic open-source monitoring application embedding a web server for serving monitoring data. Several commercial applications that are similar to ntopng are available from companies such as Boundary [26], AppNeta FlowView [33], Lancope StealthWatch [31], and Riverbed Cascade [32]. However, these applications are proprietary, often available only as a SaaS (Software as a Service) and based on the flow-paradigm (thus not fully real-time nor highly accurate) These applications are difficult to integrate with other monitoring systems because they are self-contained. Many open source network-monitoring tools are also available : packet analysers such as Wireshark [30], flow-based tools such as Vermont (VERsatile MONitoring Toolkit) [27] or YAF (Yet Another Flowmeter) [29]. Yet, 15 years after its introduction, ntopng offers singular performance, openness and ease of integration.

## 7. Final Remarks

This paper presented ntopng, an open-source, real-time traffic monitoring application. ntopng is fully scriptable by means of an embedded Lua JIT interpreter, guaranteeing both flexibility and performance. Monitoring data is represented using HTML 5 served by the embedded web server, and it can be exported to external monitoring applications by means of a REST API or through log files that can be processed by distributed log processing platforms. Validations tests have demonstrated that ntopng can effectively monitor 10 Gbit traffic on commodity hardware due to its efficient processing framework.

## 8. Code Availability

This work is distributed under the GNU GPLv3 license and is freely available in source format at the ntop home page https://svn.ntop.org/svn/ntop/trunk/ntopng/ for both Windows and Unix systems including Linux, MacOS X, and FreeBSD. The PF_RING framework used during the validation phase is available from https://svn.ntop.org/svn/ntop/trunk/PF_RING/.

## 9. Acknowledgement

Our thanks to Italian Internet domain Registry that has greatly supported the development of ntopng, Alexander Tudor <alex@ntop.org> and Filippo Fontanelli <fontanelli@ntop.org> for their help and suggestions.

## 10. References

1. P. Phaal, S. Panchen, and S. McKee, InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, RFC 3176, September 2001.
2. B. Claise, Cisco Systems NetFlow Services Export Version 9, RFC 3954, October 2004.
3. S. Leinen, Evaluation of Candidate Protocols for IP Flow Information Export (IPFIX), RFC 3955, October 2004.
4. A. Caesar, Enabling NetFlow on a vSwitch, http://www.plixer.-com/blog/network-monitoring/enabling-netflow-on-a-vswitch/, May 2013.
5. R. Lämmel, Google's MapReduce Programming Model — Revisited, Science of Computer Programming, 2008.
6. R. Bendrath, M. Müller, The end of the net as we know it? Deep packet inspection and internet governance, Journal of New Media & Society, November 2011.
7. T. Dierks, E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.1, RFC 4346, April 2006.
8. F. Fusco, M. Vlachos, X. Dimitropoulos, L. Deri, Indexing million of packets per second using GPUs, Proceedings of IMC 2013 Conference, October 2013.
9. N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, L. Jianying, NetFPGA - An Open Platform for Gigabit-Rate Network Switching and Routing, Proceeding of MSE '07 Conference, June 2007.
10. F. Fusco, L. Deri, High Speed Network Traffic Analysis with Commodity Multi-core System, Proceedings of IMC 2010 Conference, November 2010.
11. A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: The Linux Virtual Machine Monitor, Proceedings of the 2007 Ottawa Linux Symposium, July 2007.
12. P. Hintjens, ZeroMQ: Messaging for Many Applications, O'Reilly, 2013.
13. L. Deri, nProbe: an Open Source NetFlow Probe for Gigabit Networks, Proceedings of Terena TNC 2003 Conference, 2003.
14. J. A. Kreibich, S. Sanfilippo, P. Noordhuis, Redis: the Definitive Guide: Data Modelling, Caching, and Messaging, O'Reilly, 2014.
15. L. Deri, F. Fusco, Realtime MicroCloud-based Flow Aggregation for Fixed and Mobile Networks, Proceedings of TRAC 2013 workshop, July 2013.
16. D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627, 2006.
17. B. Harzog, Real-Time Monitoring: Almost Always a Lie, http://performancecriticalapps.prelert.com/articles/share/281286/, December 2013.
18. Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano, nDPI: Open-Source High-Speed Deep Packet Inspection, Proceedings of TRAC 2014 Workshop, August 2014.
19. T. Bujlow, V. Carela-Español, P. Barlet-Ros, Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification, Technical Report, Version 3, June 2013.
20. M. L. Reuven, At The Forge: Twitter Bootstrap, Linux Journal, June 2012.
21. L. Richardson, S. Ruby, RESTful Web Services, O'Reilly, 2007.
22. R. Gerhards, The Syslog Protocol, RFC 5424, March 2009.
23. 3GPP, General Packet Radio Service (GPRS); Service Description, Stage 2, Technical Specification 3GPP SP-56, V11.2.0, 2012.
24. M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, N. Feamster, Building a Dynamic Reputation System for DNS., Proceedings of USENIX Security Symposium, 2010.
25. M. Bostock, Data-Driven Documents (d3.js): a Visualization Framework for Internet Browsers Running JavaScript, http://d3js.org, .2012.
26. M. Joshi, G. Chirag, Agent Base Network Traffic Monitoring, International Journal of Innovative Research in Science, Engineering and Technology, Vol. 2, Issue 5, May 2013.
27. B. Cook, Boundary Meter 2.0 – Design, http://boundary.com/blog/2013/09/27/welcome-to-meter-2-design/, September 2013.
28. R. Lampert, et al., Vermont-A Versatile Monitoring Toolkit for IPFIX and PSAMP, Proceedings of MonAM 2006, 2006.
29. C. Inacio, B. Trammell, Yaf: yet another flowmeter, Proceedings of the 24th LISA Conference, 2010.
30. A. Orebaugh, G. Ramirez, J. Beale, Wireshark & Ethereal Network Protocol Analyzer Toolkit, Syngress, 2006.
31. Lancope Inc., StealthWatch Architecture, http://www.lancope.-com/products/stealthwatch-system/architecture/, 2014.
32. Riverbed Inc., Riverbed Cascade, http://www.riverbed.com/cascade/products/riverbed-nba.php, 2014.
33. AppNeta Inc., Flow View, http://www.appneta.com/products/flowview/, 2014.
34. Unspam Technologies Inc., Project HoneyPot, http://www.projecthoneypot.org, 2014.
35. L. Daigle, WHOIS Protocol Specification, RFC 3912, September 2004.
36. A. Newton, A Lightweight UDP Transfer Protocol for the Internet Registry Information Service, RFC 4993, August 2007.
37. S. Hollenbeck, Extensible Provisioning Protocol (EPP), RFC 4930, August 2009.
38. Draios Inc, sysdig, http://www.sysdig.org, 2014.
39. ntop, Running nProbe and ntopng on a Ubiquity EdgeRouter Lite, http://www.ntop.org/nprobe/running-nprobe-and-ntopng-on-ubiquity-edgerouter-lite/, December 2013.

# Towards Detecting Target Link Flooding Attack

*Lei Xue*[†], *Xiapu Luo*[†‡*], *Edmond W. W. Chan*[†], *and Xian Zhan*[†]
*Department of Computing, The Hong Kong Polytechnic University*[†]
*The Hong Kong Polytechnic University Shenzhen Research Institute*[‡]
{*cslxue,csxluo*}*@comp.polyu.edu.hk*, {*edmond0chan,chichoxian*}*@gmail.com*

## Abstract

A new class of target link flooding attacks (LFA) can cut off the Internet connections of a target area without being detected because they employ legitimate flows to congest selected links. Although new mechanisms for defending against LFA have been proposed, the deployment issues limit their usages since they require modifying routers. In this paper, we propose *LinkScope*, a novel system that employs both the end-to-end and the hop-by-hop network measurement techniques to capture abnormal path performance degradation for detecting LFA and then correlate the performance data and traceroute data to infer the target links or areas. Although the idea is simple, we tackle a number of challenging issues, such as conducting large-scale Internet measurement through noncooperative measurement, assessing the performance on asymmetric Internet paths, and detecting LFA. We have implemented *LinkScope* with 7174 lines of C codes and the extensive evaluation in a testbed and the Internet show that *LinkScope* can quickly detect LFA with high accuracy and low false positive rate.

***Keywords***: Network Security, Target Link Flooding Attack, Noncooperative Internet Measurement

## 1 Introduction

DDoS attacks remain one of the major threats to the Internet and recent years have witnessed a significant increase in the number and the size of DDoS attacks [1, 2], not to mention the 300 Gbps direct flooding attacks on Spamhaus and the record-breaking 400 Gbps NTP reflection attack on CloudFlare. However, it is not difficult to detect such bandwidth DDoS attacks, because the attack traffic usually reaches the victim and has difference from legitimate traffic [3].

Recent research discovered a new class of target link flooding attacks (LFA) that can effectively cut off the Internet connections of a target area (or guard area) *without* being detected [4, 5]. More precisely, an attacker first selects persistent links that connect the target area to the Internet and have high flow density, and then instructs bots to generate legitimate traffic between themselves and public servers for congesting those links [5]. If the paths among bots cover the target area, an attacker can also send traffic among themselves to clog the network [4].

It is difficult to detect LFA because (1) the target links are selected by an attacker. Since the target links may be located in an AS different from that containing the target area and the attack traffic will not reach the target area, the victim may not even know he/she is under attack; (2) each bot sends low-rate protocol-conforming traffic to public servers, thus rendering signature-based detection systems useless; (3) bots can change their traffic patterns to evade the detection based on abnormal traffic patterns. Although a few router-based approaches have been proposed to defend against such attacks [6–8], their effectiveness may be limited because they cannot be widely deployed to the Internet immediately. Note that LFA has been used by attackers to flood selected links of four major Internet exchange points in Europe and Asia [6].

Therefore, it is desirable to have a practical system that can help victims detect LFA and locate the links under attack whenever possible so that victims may ask help from upstream providers to mitigate the effect of LFA. We fill this gap by proposing and implementing a system, named *LinkScope*, which employs both end-to-end and hop-by-hop network measurement techniques to achieve this goal. The design of *LinkScope* exploits the nature of LFA including (1) it causes severe congestion on persistent links. Note that light congestion cannot disconnect the target area from the Internet; (2) although the congestion duration will be much shorter than that caused by traditional bandwidth DDoS, the congestion period caused by LFA should not be too short. Otherwise, it cannot cause severe damage to the victim; (3) to cut off the Internet connections of a target area, LFA has to

[*]The corresponding author.

continuously clog important links. Otherwise, the victim can still access the Internet. *LinkScope* actively collects samples of network path performance metrics and uses abnormal performance degradation to detect LFA.

Although the basic idea is simple, our major contributions lie in tackling a number of challenging issues to realize a practical detection system, including:

1. Since the target links are selected by an attacker, a user has to monitor as many paths as possible. However, the majority of existing network measurement systems have limited scalability because they require installing measurement tools on both ends of each path [9]. We solve this issue from two aspects. First, we design *LinkScope* as a noncooperative measurement tool that only needs the installation on one end of a path. Therefore it can cover much more paths than existing systems. Second, we strategically select important paths for measurement.

2. Due to the prevalence of asymmetric routes [10], we equip *LinkScope* with the capability to differentiate the performance metrics on the forward path (i.e., from the host where *LinkScope* is running to a remote host) and that on the reverse path. It empowers a user to infer which path(s) is under attack.

3. Although network failures may also lead to abnormal path metrics, they will not result in the same effect on all path metrics as that caused by LFA. For example, LFA will cause temporal instead of persistent congestion. By learning the normal profiles of a set of path metrics, *LinkScope* can detect LFA, differentiate it from network failures, and identify different attack patterns.

4. By conducting hop-by-hop measurement, *LinkScope* locates the target link or the target area on the forward path. Although *LinkScope* may not locate the target link on the reverse path in the absence of reverse traceroute data, we will explore possible solutions, such as reverse traceroute, looking glass, etc, in future work.

We have implemented *LinkScope* with 7174 lines of C codes and to the best of our knowledge *LinkScope* is the first system that can conduct both end-to-end and hop-by-hop noncooperative measurement. The extensive evaluations in a testbed and the Internet show that *LinkScope* can quickly detect LFA with high accuracy and low false positive rate.

The rest of this paper is organized as follows. Section 2 describes *LinkScope*'s methodology and Section 3 details the design and implementation of *LinkScope*. The evaluation results obtained from a testbed and the Internet are reported in Section 4. After introducing related work in Section 5, we conclude the paper with future work in Section 6.

## 2 Methodology

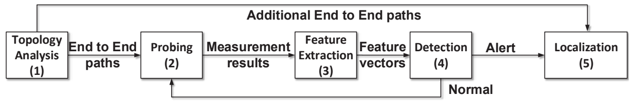


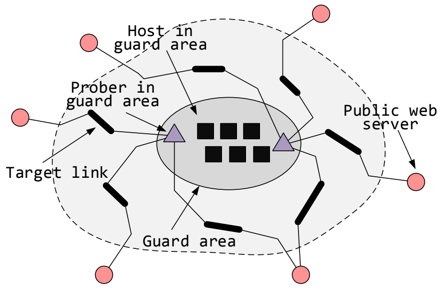
Figure 1: Major steps for detecting LFA and locating target links/areas.

Fig. 1 illustrates the major steps in our methodology for detecting LFA and locating target links/areas whenever possible. The first step, detailed in Section 2.1, involves identifying potential target links and enumerating a set of end-to-end paths that cover potential target links. Depending on the available resource, we conduct noncooperative Internet measurement on selected paths and Section 2.2 describes the measurement method and the corresponding performance metrics. Section 2.3 elaborates on the third and the fourth steps where the feature extraction algorithm turns raw measurement results into feature vectors that will be fed into the detection module for determining the existence of LFA. If there is no attack, the system will continue the measurement. Otherwise, the localization mechanism, introduced in Section 2.4, will be activated for inferring the links or areas under attack.

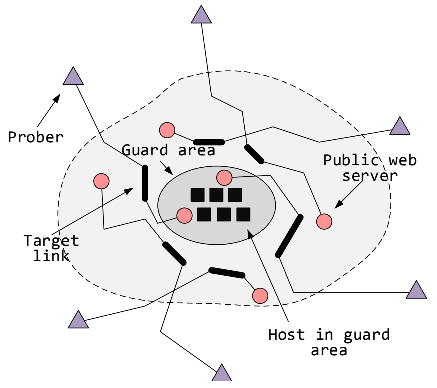
### 2.1 Topology Analysis

Adopting the noncooperative measurement approach, *LinkScope* only needs to be installed on one end of an Internet path, which is named as a prober. The current implementation of *LinkScope* can use almost any web server as the other end.

There are two common strategies to deploy *LinkScope*. Fig. 2(a) shows the first one, named self-initiated measurement, where *LinkScope* runs on hosts within the guard area. By selecting Web servers in different autonomous systems (AS), a user can measure many diverse Internet paths for covering all possible target links. The second scenario, as illustrated in Fig. 2(b), is the cloud-based measurement where *LinkScope* runs on a group hosts outside the guard area (e.g., virtual machines (VM) in different data centers) and measures the paths between themselves and hosts close to the guard area or even hosts within the guard area. Although the latter case is similar to the scenario of utilizing cooperative measurement systems that require the control of both ends



(a) Self-initiated measurement.



(b) Cloud-based measurement.

Figure 2: Deployment strategies of *LinkScope*.

of a path, using *LinkScope* can simplify the deployment, because only one end needs to install *LinkScope*. By running *LinkScope* on hosts in diverse networks and/or selecting web servers in various locations, the paths under measurement may include all possible target links.

Given a guard area, we first construct the network topology between it and its upstream ASes by performing paris-traceroute [11] from a group of hosts (e.g., VM in clouds or looking glasses [12] ) to web servers close to or within the guard area, or using systems like Rocketfuel [13]. From the topology, we can identify potential target links following the LFA's strategy that selects persistent links with high flow density [5]. The flow density of a link is defined as the number of Internet paths between bots and public servers in the target area, which include that link.

Given a set of potential target links denoted as $L = \{l_1, l_2, ..., l_M\}$, we select a set of paths for measurement, which is indicated by $P = \{p_1, p_2, ..., p_N\}$. Since there may be more than one path traversing certain target links, we define three rules to guide the path selection:

- For the ease of locating target links, paths that contain one target link will be selected.
- The number of paths sharing the same remote host should be minimized to avoid mutual interference. It is desirable that each path has different remote host.
- Similar to the second rule, the number of paths initialized by one prober should be minimized to avoid self-induced congestion.

## 2.2 Measurement approaches

As LFA will congest the selected links, it will lead to anomalies in the following path performance metrics, including,

- Packet loss rate, which will increase because the link is clogged;
- Round-trip time (RTT), which may also increase because of the full queue in routers under attack;
- Jitter, which may have large variations when bots send intermittent bursts of packets to congest the link [14], thus leading to variations in the queue length;
- Number of loss pairs [15], which may increase as a pair of probing packets may often see full queues due to LFA;
- Available bandwidth, which will decrease because the target link is congested;
- Packet reordering, which may increase if the router under attack transmits packets through different routes;
- Connection failure rate, which may increase if the target area has been isolated from the Internet due to severe and continuous LFA.

Besides measuring the above metrics, *LinkScope* should also support the following features:

- Conduct the measurements within a legitimate TCP connection to avoid the biases or noises due to network elements that process TCP/UDP packets in a different manner and/or discard all but TCP packets belonging to valid TCP connections.
- Perform both end-to-end and hop-by-hop measurements. The former can quickly detect the anomalies caused by LFA while the latter facilitates localizing the target links/areas.
- Support the measurement of one-way path metrics because of the prevalence of asymmetric routing.

To fulfill these requirements, *LinkScope* contains the following three probing patterns:

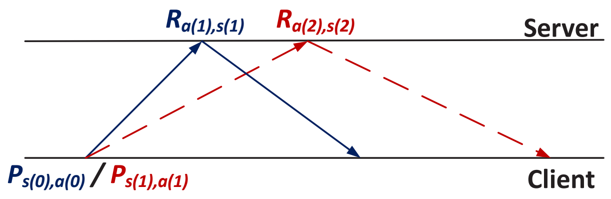


Figure 3: Round trip probing (RTP) pattern.

### 2.2.1 Round Trip Probing (RTP)

We proposed the Round Trip Probing (RTP) pattern to measure RTT, one-way packet loss, and one-way packet reordering in [16]. As shown in Fig. 3, each RTP measurement involves sending two back-to-back probing packets (i.e., $P_{s(0),a(0)}$ and $P_{s(1),a(1)}$) with customized TCP sequence number (i.e., s(0),s(1)) and acknowledgement number (i.e., a(0) and a(1)) to the remote host. The advertising window of each probing packet is set to 2 maximal segment size (MSS) and therefore each probing packet will elicit one response packet (i.e., $R_{a(1),s(1)}$ and $R_{a(2),s(2)}$). By analyzing the sequence numbers and the acknowledgement numbers in the response packets, we can decide whether there is packet loss/packet reordering occurred on the forward path or the reverse path. If the server supports TCP options like timestamp or SACK, they can ease the detection of forward path packet loss [16]. Moreover, RTT can be measured as the duration from sending $P_{s(0),a(0)}$ to receiving $R_{a(1),s(1)}$.

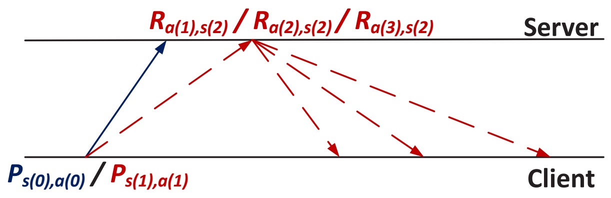

$R_{a(1),s(2)}$ / $R_{a(2),s(2)}$ / $R_{a(3),s(2)}$
Server
$P_{s(0),a(0)}$ / $P_{s(1),a(1)}$
Client

Figure 4: Extended two way probing (eTWP) pattern with $w = 3$.

### 2.2.2 Extended Two Way Probing (eTWP)

We proposed the original Two Way Probing (TWP) pattern for measuring one-way capacity in [17]. The extended Two Way Probing (eTWP) pattern has similar probing packets as that of TWP. The difference is that eTWP will induce more response packets from the remote host than TWP does. As shown in Fig. 4, TWP(or eTWP) involves sending two back-to-back probing packets (i.e., $P_{s(0),a(0)}$ and $P_{s(1),a(1)}$). The first probing packet uses zero advertising window to prevent the server from sending back responses on the arrival of $P_{s(0),a(0)}$. In TWP, the advertising window in $P_{s(1),a(1)}$ is equal to 2 MSS so that it will trigger two packets from the server [17]. Since a packet train can characterize more loss patterns than a packet pair [18], we enlarge the advertising window in $P_{s(1),a(1)}$ from 2 to $w$ ($w > 2$) in eTWP. Note that increasing $w$ requires *LinkScope* to handle more patterns of response packets.

As the server may dispatch $w$ packets back-to-back if its congestion window allows, we can compute the time gap between the first and the $w$-th packet, denoted as $G_r$, and define $\theta_r$ to characterize the available bandwidth on the reverse path.

$$\theta_r = \frac{MSS \times (w-1)}{G_r}. \tag{1}$$

Note that $\theta_r$ may not be equal to the real available bandwidth [19] but its reduction could indicate congestion [20].

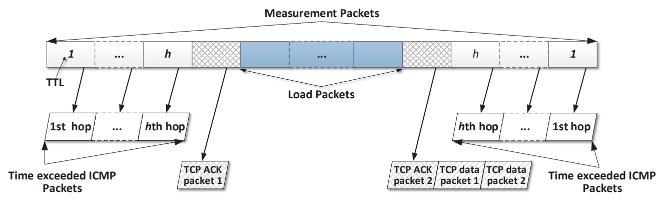
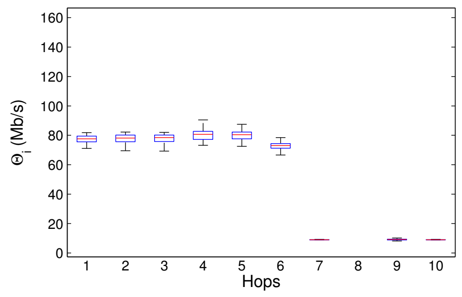


Figure 5: Modified recursive packet train (RPT) pattern.

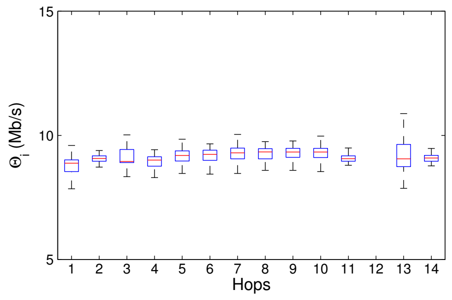
### 2.2.3 Modified Recursive Packet Train (mRPT)

Hu et al. proposed the original recursive packet train (RPT), which was employed in Pathneck for detecting the location of a network path's bottleneck [20]. The original RPT consists of a group of load packets and a set of TTL-limited measurement packets and Pathneck uses UDP packets to construct RPT. We modify RPT to support end-to-end and hop-by-hop measurements in a TCP connection and remove redundant packets. Fig.5 illustrates the modified RPT, denoted as mRPT, where each rectangle is a probing packet and each parallelogram indicates a response packet triggered by a probing packet. mRPT has $h$ pairs of small measurement packets, whose TTL values are equal to the number in those rectangles. Since a router will send back a time exceeded ICMP packet when a packet's TTL becomes zero, a pair of ICMP packets will be sent back after mRPT passes through a router. We use $G_{I(i)}$ to denote the time gap between the two ICMP packets from the $i$-th hop. *LinkScope* does not use a fixed number of measurement packets (e.g., 30 in Pathneck [20]) because we do not want them to reach the server and LFA usually targets on links outside the victim's network. Instead, *LinkScope* first determines $h$ by doing a traceroute.

The load packets are customized TCP packets that belong to an established TCP connection and carry an invalid checksum value or a TCP sequence number so that they will be discarded by the server. There are two special packets (i.e., R1 and R2) between the load packets sand the measurement packets. They have the same size as the load packets and work together to accomplish two tasks: (1) each packet triggers the server to send back a TCP ACK packet so that the prober can use the time gap between these two ACK packets, denoted as $G_A$, to estimate the interval between the head and tail load packet; (2) induce two TCP data packets from the server to start the measurement through RTP [16]. To achieve these goals, *LinkScope* prepares a long HTTP request whose length is equal to two load packets and puts half of it to R1 and the remaining part to R2. To force the server to immediately send back an ACK packet on the arrival of R1 and R2, we first send R2 and then R1, because a TCP server will send back an ACK packet when it receives an out-of-order TCP segment or a segment that fills a gap in the sequence space [21].



(a) Path from Korea to Hong Kong



(b) Path from Taiwan to Hong Kong

Figure 6: $\theta_i$ measured on two paths to Hong Kong.

To characterize the per-hop available bandwidth and end-to-end available bandwidth, *LinkScope* defines $\theta_i$ (i=1,...,h) and $\theta_e$ as follows:

$$\theta_i = \frac{S_L \times (N_L + 2) + S_M \times (h - i)}{G_{I(i)}}, i = 1, \ldots, h, \quad (2)$$

$$\theta_e = \frac{S_L \times N_L}{G_A}, \quad (3)$$

where $S_L$ and $S_M$ denote the size of a load packet and that of a measurement packet, respectively. $N_L$ is the number of load packets.

Note that since the packet train structure cannot be controlled after each hop, similar to $\theta_r$, $\theta_i$ (or $\theta_e$) may not be an accurate estimate of per-hop available bandwidth (or end-to-end available bandwidth) but their large decrement indicates serious congestion [20]. Since LFA will lead to severe congestion on target links, $\theta_i$ of the target link or $\theta_e$ on the path covering the target link will be throttled.

Fig.6 shows $\theta_i$ on two paths to a web server in our campus, whose last four hops are located in the campus network. Since the last but two hops did not send back ICMP packets, there is no $\theta_i$ on that hop. On the path from Korea to Hong Kong, $\theta_i$ drops from around 80Mbps to around 9Mbps on the 7th hop. It is because the bandwidth of each host in campus network is limited to 10Mbps. On the path from Taiwan to Hong Kong, $\theta_i$ is always around 9Mbps. It may be due to the fact the first hop's available bandwidth is around 9Mbps.

## 2.3 Anomaly detection

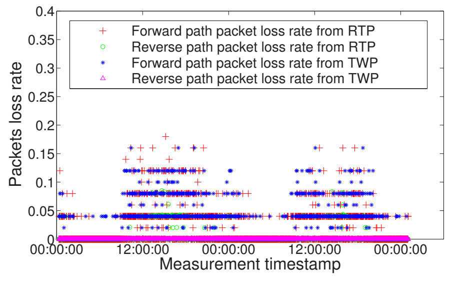
Table 1: Detail metrics measured during one probe.

| Direction | Metric | Defination |
|---|---|---|
| *Forward* | $\theta_e$ | Characterizing available bandwidth through mRPT. |
| | $R_{RFPL}$ | Packet loss rate from RTP. |
| | $R_{TFPL}$ | Packet loss rate from eTWP. |
| | $R_{RFPL2}$ | Loss pair rate from RTP. |
| | $R_{TFPL2}$ | Loss pair rate from eTWP. |
| | $R_{RFPR}$ | Packet reordering rate from RTP. |
| | $R_{TFPR}$ | Packet reordering rate from eTWP. |
| *Reverse* | $\theta_r$ | Characterizing available bandwidth through eTWP. |
| | $R_{RRPL}$ | Packet loss rate from RTP. |
| | $R_{TRPL}$ | Packet loss rate from eTWP. |
| | $R_{RRPL2}$ | Loss pair rate from RTP. |
| | $R_{TRPL2}$ | Loss pair rate from eTWP. |
| | $R_{RRPR}$ | Packet reordering rate from RTP. |
| | $R_{TRPR}$ | Packet reordering rate from eTWP. |
| $Round-trip$ | $RTT$ | Round-trip time. |
| | $J_{RTT}$ | Round-trip time variation (jitter). |
| | $Fail_{RTP}$ | Connection failure rate in RTP. |
| | $Fail_{TWP}$ | Connection failure rate in eTWP. |

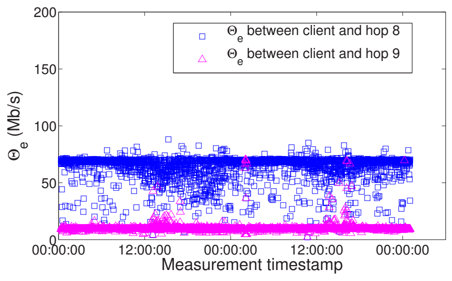
We define two metric vectors in Eqn. (4) and (5), which cover selected performance metrics, for the forward path and the reverse path, respectively. Table 1 lists the meaning of each performance metric.

$$\overrightarrow{F_{forward}} = \{\theta_e, R_{RFPL}, R_{TFPL}, R_{RFPL2}, R_{TFPL2}, R_{RFPR}, R_{TFPR}, RTT, J_{RTT}, Fail_{RTP}, Fail_{TWP}\}^{\mathrm{T}} \tag{4}$$

$$\overrightarrow{F_{reverse}} = \{\theta_r, R_{RRPL}, R_{TRPL}, R_{RRPL2}, R_{TRPL2}, R_{RRPR}, R_{TRPR}, RTT, J_{RTT}, Fail_{RTP}, Fail_{TWP}\}^{\mathrm{T}} \tag{5}$$



(a) Packet loss rate.



(b) $\theta_e$

Figure 7: Performance of a path from Japan to Hong Kong over 48 hours.

*LinkScope* keeps collecting samples of these metrics and builds a normal profile for each path using the data in the absence of LFA. Since the measurement results show a diurnal pattern, we build the normal profile for each or several hours per day. For example, Fig. 7(a) shows the diurnal pattern of forward path packet loss rate and $\theta_e$ on a path from Japan to Hong Kong over 48 hours.

Then, *LinkScope* uses the Mahalanobis distance [22] to quantify the difference between the profile and a new round of measurement results as follows:

$$D_M(\overrightarrow{F}) = \sqrt{(\overrightarrow{F} - \overrightarrow{\lambda})^{\mathrm{T}}\Omega^{-1}(\overrightarrow{F} - \overrightarrow{\lambda})}, \tag{6}$$

where $\overrightarrow{F}$ is the metric vector from a round of measurement results described in Section 3. $\overline{\overrightarrow{\lambda}}$ denotes the mean metric vector in the profile and $\Omega$ is the covariance matrix .

$$\Omega = \frac{1}{n-1}\sum_{i=1}^{n}(\lambda_i - \bar{\lambda})(\lambda_i - \bar{\lambda})^T, \tag{7}$$

where $\lambda_i$ is the $i$-th metric in the profile, $n$ is the number of metrics and

$$\bar{\lambda} = \frac{1}{n}\sum_{i=1}^{n}\lambda_i. \tag{8}$$

Finally, *LinkScope* employs the non-parametric cumulative sum (CUSUM) algorithm [23] to capture the abrupt changes in the Mahalanobis distance (i.e., $D_M$). The non-parametric CUSUM algorithm assumes that the average distance is negative in normal situation and becomes positive when path is under attack. We use $D_n$ to denote the distance measured in $n$-th probe and turn $\{D_n\}$ into a new sequence $\{X_n\}$ through

$$X_n = D_n - \overline{D_n}, \tag{9}$$
$$\overline{D_n} = mean(D_n) + \alpha std(D_n), \tag{10}$$

where $\alpha$ is an adjustable parameter, $mean(D_n)$ is the mean value of $D_n$, and $std(D_n)$ is the standard deviation of $D_n$. The non-parametric CUSUM algorithm defines a new sequence $\{Y_n\}$ by Eqn. (11).

$$Y_n = \begin{cases}(Y_{n-1}+X_n)^+, n>0,\\ 0, \quad n=0,\end{cases} \quad where \quad x^+ = \begin{cases}x, & x>0,\\ 0, & otherwise.\end{cases} \tag{11}$$

Since the Mahalanobis distance quantifies the difference between the profile and a new observation, a measurement result showing better network performance may also be regarded as anomalies. To remedy this problem, we only consider the alerts where the measured performance metrics become worse than the normal profile (e.g. smaller $\theta_e$ and larger packet loss rate) because of the nature of LFA.

## 2.4 Locating the target link

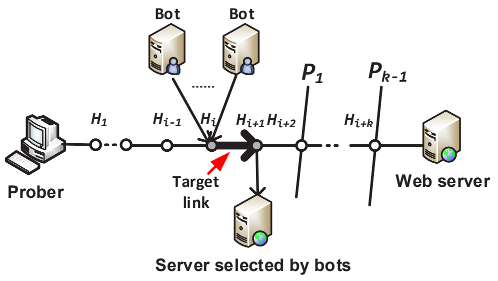


Figure 8: Locating the target links.

When performance anomaly is detected on a forward path, *LinkScope* tries to locate the target link through two steps. We use an example shown in Fig.8 to illustrate the steps, where bots send traffic to the server selected by bots in order to congest the link between $H_i$ and $H_{i+1}$.

First, based on the hop-by-hop measurement results from mRPT, *LinkScope* knows that the path from $H_1$ to $H_{i-1}$ is not under attack. Second, according to the topology analysis, *LinkScope* will perform measurement on other paths that cover the hops after $H_i$, such as $P_1$ going through $H_{i+1}$ and $P_{k-1}$ covering $H_{i+k}$. If one new path (e.g. the one covering $H_{i+j}$) does not have poor performance like the original path, then the target link is in the area from $H_i$ to $H_{i+j-1}$. The rational behind this approach comes from the nature of LFA that congests a selected link so that all paths including that link will suffer from similar performance degradation. By contrast, other paths will not have similar patterns.

Since the paths identified in Section 2.1 may not cover all hops on the original path, we propose the following steps to look for new paths.

1. For a hop, $H_k$, we utilize high-speed scanning tools such as Zmap [24] to look for web servers in the same subnet as $H_k$, which can be determined through systems like traceNET [25]. If a web server is found, *LinkScope* performs traceroute to this web server and checks whether the path to the server goes through $H_k$.

2. We look for web servers located in the same AS as $H_k$ and then check whether the paths to those web servers go through $H_k$.

3. We look for web servers located in the buddy prefix [26]as $H_k$ and then check whether the paths to those web servers go through $H_k$.

4. If no such path can be found, we check next hop.

We acknowledge that this method may not be applied to reverse paths because it is difficult to get the traceroute on the reverse path (i.e., from the remote host to the prober). In future work, we will explore two possible approaches to tackle this issue. First, the victim may combine both self-initiated measurement and cloud-based measurement using hosts under control after anomalies are detected. Second, using reverse traceroute [27] or looking glass [28] may be able to obtain the traceroute on the reverse path.

## 3 *LinkScope*

In this section, we detail the design of *LinkScope* whose architecture is illustrated in Fig. 9. We have implemented *LinkScope* with 7174 lines of C codes and the extensive evaluation results obtained in a testbed and the Internet are reported in Section 4.

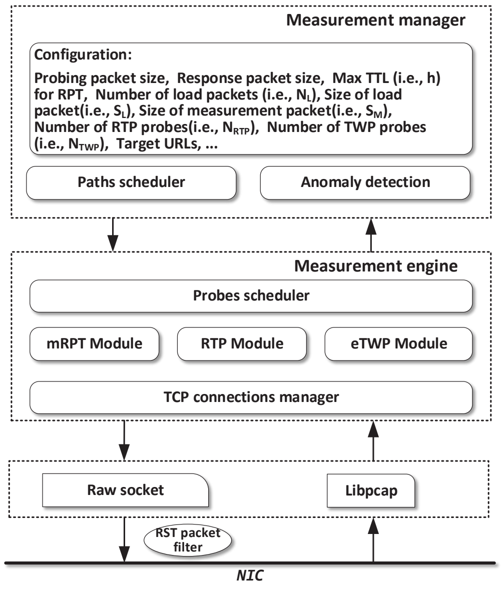

Measurement manager
Configuration:
Probing packet size, Response packet size, Max TTL (i.e., h) for RPT, Number of load packets (i.e., N_L), Size of load packet(i.e., S_L), Size of measurement packet(i.e., S_M), Number of RTP probes(i.e., N_RTP), Number of TWP probes (i.e., N_TWP), Target URLs, ...
Paths scheduler
Anomaly detection
Measurement engine
Probes scheduler
mRPT Module
RTP Module
eTWP Module
TCP connections manager
Raw socket
Libpcap
RST packet filter
NIC

Figure 9: The architecture of *LinkScope*.

### 3.1 Measurement Manager

The original designs of RTP, TWP, and RPT are not limited to specific application layer protocol. We use HTTP as the driving protocol because tremendous number of web servers are publicly available for the measurement. In future work, we will explore other protocols.

We also realize a tool named *WebChecker* to collect basic information about the path and the remote server. It runs Paris-traceroute [11] to determine the number of hops between a prober and the server, and then sets $h$ so that the measurement packet in mRPT can reach the network perimeter of the server.

*WebChecker* also enumerates suitable web objects in a web server and output a set of URLs. It prefers to fetching static web objects (e.g., figure, pdf, etc.) starting from the front page of a web site and regards a web object as a suitable one if its size is not less than 10K bytes. Furthermore, similar to TBIT [29], *WebChecker* will check whether the web server supports TCP options, including Timestamp, Selective Acknowledgment(SACK), and HTTP header options such as Range [30]. These options may simplify the process of *LinkScope* and enhance its capability. For example, if the server supports MSS, *LinkScope* can control the size of response packets. Supporting Timestamp and SACK can ease the detection of forward path packet loss [16].

The paths scheduler in *LinkScope* manages a set of probing processes, each of which conducts the measurement for a path. To avoid self-induced congestion, the path scheduler will determine when the measurement for a certain path will be launched and how long a path will be measured. Currently, each path will be measured for 10 minutes. The probing packet size, the response packet size, and the load packet size are set to 1500 bytes. The number of load packets is 20 and the size of measurement packet is 60 bytes. The number of RTP probes and the number of TWP probes are equal to 30. All these parameters can be configured by a user.

The collected measurement results will be sent to the anomaly detection module for detecting LFA.

## 3.2 Measurement Engine

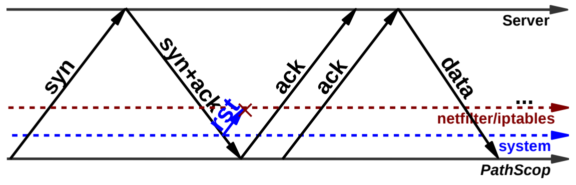
In the measurement engine, the probes scheduler manages the measurements on a path. A round of measurement consists of one probe based on the mRPT pattern, $N_{RTP}$ probes based on the RTP pattern, and $N_{TWP}$ probes based on the eTWP pattern. A probe consists of sending the probing packets and processing the response packets. After finishing a round of measurement, the probes scheduler will deliver the parsed measurement results to the anomaly detection module and schedule a new round of measurement.

The mRPT, RTP, and eTWP modules are in charge of preparing the probing packets and handling the response packets according to the corresponding patterns. Before conducting measurement based on mRPT, *LinkScope* sets each measurement packet's IPID to its TTL. Since each pair of measurement packets will trigger two ICMP packets, *LinkScope* inspects the ICMP packet's payload, which contains the IP header and the first 8 bytes of the original packet's data, for matching it to the measurement packet.

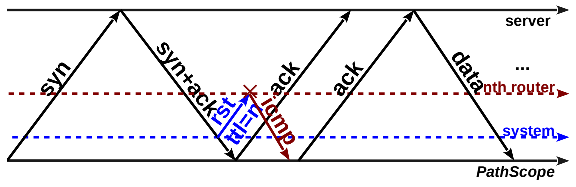
It is worth noting that in each round of measurement for a path all probes are performed within *one* TCP connection. Such approach can mitigate the negative effect due to firewall and instable routes, because stateful firewall will drop packets that do not belong to any established TCP connection and load balancer may employ the five tuple of <src IP, src Port, dst IP, dst Port, Protocol> to select routes.

The TCP connections manager will establish and maintain TCP connections. If the server supports TCP options like MSS, Timestamp, and SACK, the TCP connections manager will use MSS option to control the size of response packet (i.e., the server will use the minimal value between its MSS and the MSS announced by *LinkScope*). It will also put the SACK-permitted option and TCP timestamp option into the TCP SYN packet sent from *LinkScope* to the server.

Since *LinkScope* needs to control the establishment of TCP connections and customize probing packets (e.g., sequence number, acknowledgement number, advertising window), all packets are sent through raw socket. Moreover, *LinkScope* uses the libpcap library to capture all response packets and then parses them for computing performance metrics.



(a) Use netfilter/iptables.



(b) Modify TTL of RST packet

Figure 10: RST packet filter.

## 3.3 RST packet filter

Since *LinkScope* constructs all packets by itself and sends them out through raw socket, OS does not know how to handle the response packets and therefore it will reply with an RST packet to the server to close the TCP connections. We employ two approaches to filter out RST packets generated by OS.

As shown in Fig. 10(a), if the system supports netfilter/iptables [31], we use it to drop all RST packets except those generated by *LinkScope*. We differentiate the RST packets from OS and that from *LinkScope* through the IPID value in the IP header because *LinkScope* will set the IPID value of its RST packets to a special value.

Since some hosts do not support netfilter/iptables, such as those Planetlab nodes [32], we propose another method as shown in Fig. 10(b). *LinkScope* first establishes a TCP connection with the web server using stream socket (i.e., SOCK_STREAM), and then uses the function *setsockopt* to set the TTL value in each packet generated by OS to a small value so that it will not reach the web server. Moreover, *LinkScope* utilizes the libpcap library to capture the TCP three-way handshaking packets generated by OS to record the initial sequence numbers selected by the local host and the web server along with other information related to the TCP connection, such as source port, and TCP options. After that, *LinkScope* will create and send probing packets through raw socket with the help of such information.

## 4 Evaluation

We carry out extensive experiments in a test-bed and the Internet to evaluate *LinkScope*'s functionality and overhead.

### 4.1 Test bed

Fig. 11 shows the topology of our test bed that connects to the Internet through the campus network. All hosts run Ubuntu system. Host 1 and Host 2 act as attackers and the public server used by attackers, respectively. D-ITG [33] is used to generate traffic for congesting the MikroTik router in red circle. The router serves as the bottleneck with 10Mbps bandwidth. Host 3 is a bridge for emulating packet loss and packet reordering and Host 4 is an NAT-enable router providing port forwarding in order to connect the web server and the LAN to the Internet. In our experiment, LAN denotes the guard area and the web server is a public server that can be accessed by nodes in the Internet. We deploy *LinkScope* on Planetlab nodes and Amazon EC2 instances.

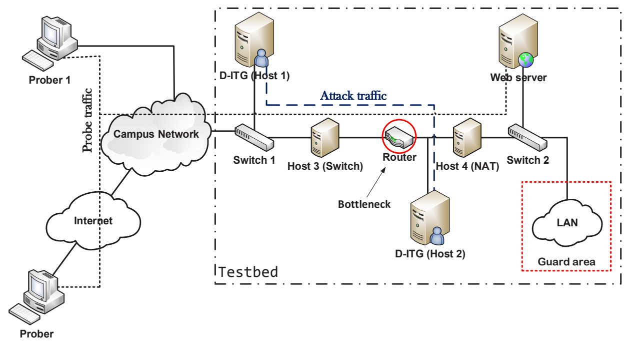


Figure 11: The topology of the testbed.

### 4.2 Emulated Attacks in the Test bed

To demonstrate that *LinkScope* can capture different kinds of LFA, we emulate four types of LFA in the testbed and use the abnormal changes in $\theta_e$ to illustrate the diverse effect due to different attacks. If the attacker floods the bottleneck with high-volume traffic, all TCP connections including the one for measurement are disconnected and $\theta_e$ becomes zero all the time. Therefore, we did not show it.

Fig. 12(a) shows $\theta_e$ under pulsing LFA where the attacker transmits high-volume bursts of traffic to congest the bottleneck [14]. The attack traffic rate is 1600 packets per second and the packet size is uniformly distributed in the range of [600, 1400] bytes. In the absence of attack, $\theta_e$ is close to the available bandwidth. Under the attack, since the bottleneck is severely congested and all connections are broken, $\theta_e$ becomes zero.

Fig.12(b) illustrates $\theta_e$ under LFA with two attack traffic rates: 400 packets per second and 800 packets per second. An attacker may change the attack traffic rate for evading the detection. We can see that when the attack rate decreases (or increases), $\theta_e$ increases (or decreases), meaning that it can capture the changes in the attack traffic rate.

Fig.12(c) represents $\theta_e$ under gradual LFA where the attack traffic rate increases from zero to a value larger than the capacity of the bottleneck. It emulates the scenario of DDoS attacks in Internet where the traffic sent from different bots may not reach the bottleneck simultaneously, thus showing the gradual increase in the attack traffic rate. Although the TCP connection for measurement was broken when the attack traffic rate almost reached its maximal value, the decreasing trend of $\theta_e$ can be employed to raise an early alarm.

Fig.12(d) demonstrates $\theta_e$ when a network element randomly drops packets. It may be due to occasional congestion or the use of random early drop (RED) in routers. We can see that although $\theta_e$ varies its values are still close to the available bandwidth.

Since LFA will cause severe intermittent congestion on target links in order to cut off the Internet connections of the guard area, we can use different patterns in performance metrics to distinguish it from other scenarios, such as long-term flooding and cable cut which will disable the Internet connection for quite a long period of time, and even identify different types of attacks, as demonstrated in Fig. 12.

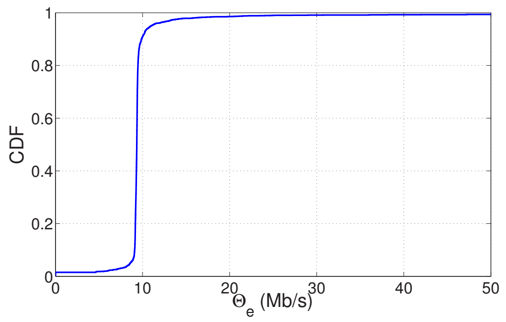
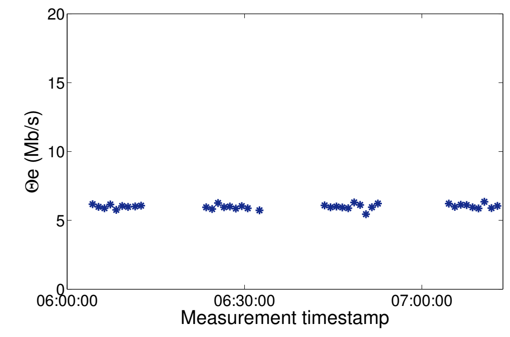


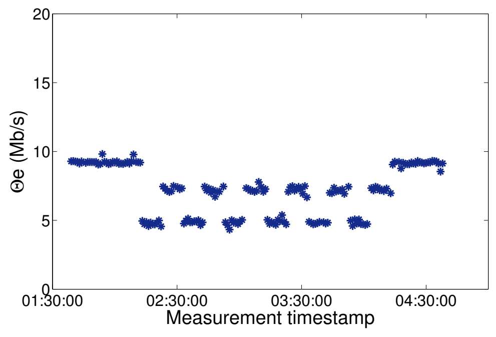
Figure 14: CDF of $\theta_e$ on path from Amsterdam to Hong Kong.

### 4.3 Internet Probing

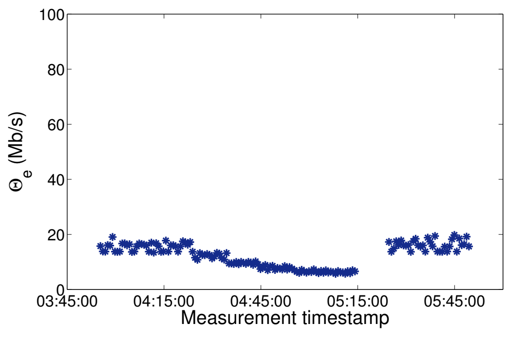
To evaluate the capability and the stability of *LinkScope*, we run it on Planetlab nodes to measure paths to Hong Kong for two days and paths to Taiwan for seven days.



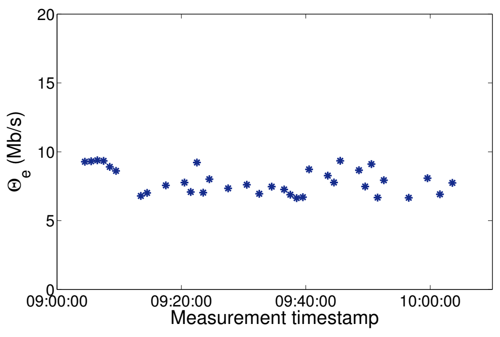
(a) Pulsing LFA.



(b) LFA with variant attack traffic rates.



(c) Gradual LFA.



(d) Random packet drop.

Figure 12: Available bandwidth measured with different attacks from Prober 1 to testbed.

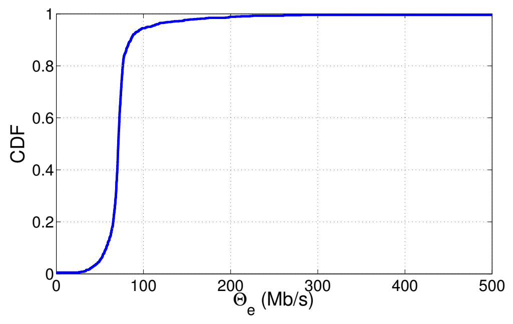


Figure 16: CDF of $\theta_e$ on path from Santa Barbara to Taipei.

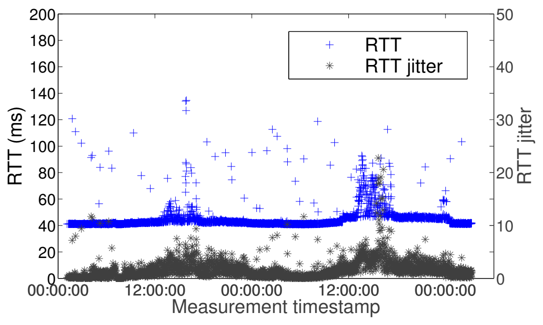
Fig.13 shows the performance metrics measured on the path from Amsterdam to Hong Kong for two days. It demonstrates the diurnal patterns in forward path/reverse path packet loss, RTT, and jitter. The path performance is better and more stable in the period from 00:00 to 12:00 than that during the period from 12:00 to 24:00. The increased loss rate may affect the measurement of $\theta_e$ as some measurement results deviate during the period from 12:00 to 24:00 as shown in Fig. 13(d). Fig.14 illustrates the CDF of $\theta_e$ on the path from Amsterdam to Hong Kong, where $\theta_e$ concentrates on 9 Mb/s.

Fig.15 demonstrates the performance metrics measured on the path from Santa Barbara (US) to Taipei for seven days. This path has stable good performance. For example, RTT is around 150ms and the jitter is less than 10 shown Fig. 15(a). The loss rate is less than 2% and there is no packet reordering. The estimated end-to-end $\theta_e$ is around 75Mbps as illustrated in Fig. 15(d) and Fig.16. Since LFA will cause severe congestion during a short duration, it will cause obvious abrupt changes in the performance metrics and get caught by *LinkScope*.

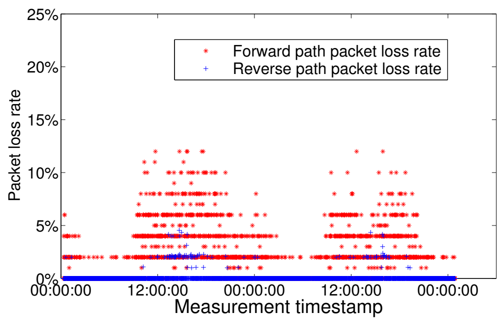
## 4.4 Detection Performance

We first evaluate *LinkScope*'s false positive rate using Internet measurement results on different paths, and then assess its detection rate using emulated attacks in our test bed.

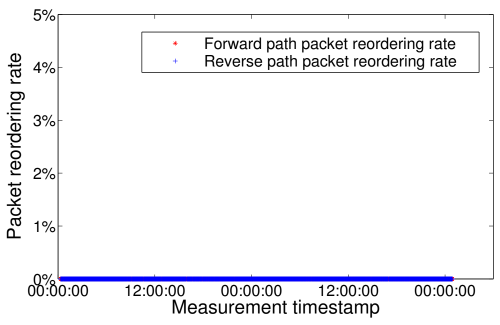
On the paths to Hong Kong, *LinkScope* conducts measurement once per minute for two days (48 hours). We divide the one-day data (24 hours) into 24 sets (one set per hour), because features are changing over time. We use the data obtained in the first day as the training data and use the remaining data to evaluate *LinkScope*'s false positive rate. Table 2 lists the false positive rates on eight paths to Hong Kong with different $\alpha$. The first four probers are Amazon EC2 VM and the last four are



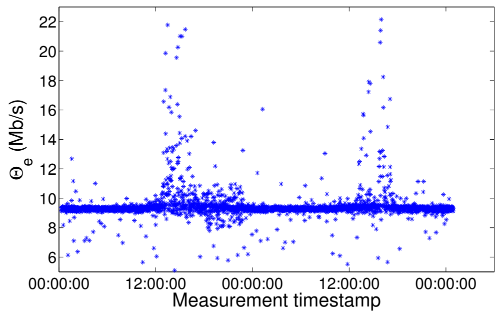
(a) RTT and RTT jitter.



(b) Packet loss rate.



(c) Packet reordering rate.



(d) $\theta_e$.

Figure 13: Performance metrics measured on the path from Amsterdam to Hong Kong for two days.

Planetlab nodes. In this experiment, we divide one-day data equally into 24 segments. The false positive rates are all less than 10% and it decreases when $\alpha$ increases, because $\alpha$ serves as a threshold and a larger $\alpha$ may cover more normal scenarios. Moreover, all false positive rates on the path from South Carolina (PL node) to Hong Kong are 0, because the performances of all metrics are very stable during both days. Table 2 shows that all false positive rates are smaller than 6% when $\alpha$ is not less than 30.

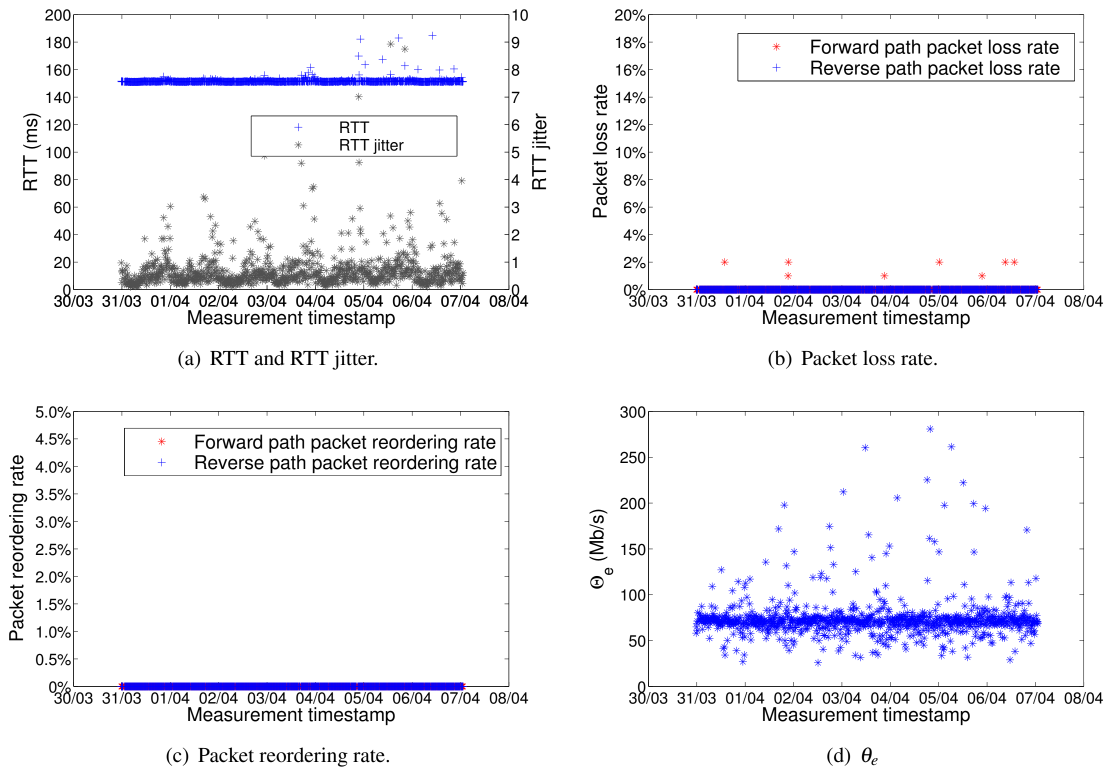
Table 3 shows false positive rate on the paths from five Planetlab nodes and two Amazon EC2 hosts to Taiwan. On these paths, *LinkScope* conducts measurement once per ten minutes for seven days. We also take the data in the first day as the training data and the remaining data for evaluation. Table 3 shows that the increases of $\alpha$ can decrease the false positive rate, except the path from US West to So-net Entertainment where no false positive detected, as a result of the stable performances during the two days.

By inspecting false positive cases, we find that almost all the false positives are due to connection failure. It may happen even without attack. Take the path from Tokyo to Hong Kong as an example, the connection failure rate in two days is 4.06%. This rate varies over time, such as, 0.90% during the period from 00:00 to 12:00 and 7.5% for the period from 12:00-24:00, because the network performance is much more unstable from 12:00 to 00:00 (such as shown in Fig.13). However, in the absence of LFA, the connection failures scatters over time while the connection failures appear continuously in the presence of LFA.

Table 4: Detection rate.

| Training data | path | $\alpha = 10$ | $\alpha = 20$ | $\alpha = 30$ |
|---|---|---|---|---|
| 20 probes | path 1 | 100.0% | 100.0% | 100.0% |
| 20 probes | path 2 | 100.0% | 100.0% | 100.0% |
| 40 probes | path 1 | 100.0% | 100.0% | 100.0% |
| 40 probes | path 2 | 100.0% | 100.0% | 100.0% |

To evaluate *LinkScope*'s detection rate, we emulate different attacks between Host 1 and Host 2 as shown Fig.11. During the pulsing LFA and gradual LFA, the detection rate are always 100%. Because when the attack traffic rate is much higher than the available bandwidth, the path is congested and none response packets can be received from the destination all the time. Table 4 lists the detection rates when the attack traffic rate is a little higher than the bandwidth (1.2 times of bandwidth). In this case, *LinkScope* can still receive some response packets and compute the measurement results. Table 4 shows that the anomaly detection rates are still



(a) RTT and RTT jitter.

(b) Packet loss rate.

(c) Packet reordering rate.

(d) $\theta_e$

Figure 15: Performance metrics measured on the path from Santa Barbara to Taipei for seven days.

Table 2: False positive rate on paths to Hong Kong.

| Prober type | Path | $\alpha = 10$ | $\alpha = 20$ | $\alpha = 30$ | $\alpha = 40$ | $\alpha = 50$ |
|---|---|---|---|---|---|---|
| EC2 | Virginia - Hong Kong | 6.32% | 5.99% | 5.12% | 4.33% | 3.67% |
| EC2 | Tokyo - Hong Kong | 5.88% | 4.02% | 2.85% | 2.07% | 1.94% |
| EC2 | Ireland - Hong Kong | 8.69% | 7.24 | 5.75% | 5.23% | 4.58% |
| EC2 | Sao Paulo - Hong Kong | 5.80% | 2.90% | 2.52% | 1.55% | 1.16% |
| PL node | Tokyo - Hong Kong | 2.19% | 1.19% | 0.60% | 0.60% | 0.60% |
| PL node | Amsterdam - Hong Kong | 4.18% | 2.61% | 1.69% | 1.30% | 0.91% |
| PL node | Beijing - Hong Kong | 3.54% | 2.96% | 2.06% | 2.67% | 1.28 |
| PL node | South Carolina - Hong Kong | 0 | 0 | 0 | 0 | 0 |

Table 3: False positive rate on paths to Taiwan with different configures.

| Prober type | Path | $\alpha = 20$ | $\alpha = 30$ | $\alpha = 40$ |
|---|---|---|---|---|
| PL node | Boston - Taipei | 2.17% | 1.45% | 0.97% |
| PL node | Urbana - Taipei | 2.18% | 1.69% | 1.45% |
| PL node | Turkey - Taipei | 2.20% | 2.19% | 1.21% |
| PL node | Tokyo - Taipei | 1.59% | 3.17% | 3.17% |
| PL node | Blacksburg - Taipei | 1.76% | 1.25% | 1.00% |
| PL node | Tokyo - ChungHwa Telecom | 2.32% | 1.45% | 1.01% |
| EC2 | Sao Paulo - Taichung | 6.56% | 4.01% | 2.15% |
| EC2 | US West - So-net Entertainment | 0 | 0 | 0 |

100% though the attacks cannot fully clog the bottleneck.

## 4.5 System load

To evaluate the system load introduced by *LinkScope*, we use htop [34] to measure the client's and web serv-

Table 5: The CPU utilizations and Load average in the probing client and web server during measurement.

| Probing processes | Measurement rate (Hz) | Probing client | | Web sever | |
|---|---|---|---|---|---|
| | | Load average | CPU utilization | Load average | CPU utilization |
| 0 | 0 | 0.01 | 0.3% | 0.00 | 0.5% |
| 1 | 2 | 0.06 | 0.3% | 0.00 | 0.5% |
| 1 | 10 | 0.10 | 0.3% | 0.01 | 0.6% |
| 2 | 10 | 0.10 | 0.4% | 0.01 | 0.6% |
| 10 | 10 | 0.11 | 1.7% | 0.02 | 0.7% |
| 50 | 10 | 0.23 | 2.4% | 0.08 | 0.8% |
| 100 | 10 | 0.47 | 2.7% | 0.09 | 0.8 % |

er's average load and average CPU utilization when *LinkScope* runs with different configurations. The client, running Ubuntu 12.04 LTS system, is equipped with Intel 3.4 GHz i7-4770 CPU, 16G memory, and 1Gbps NIC, and the web server is equipped with Intel 2.83 GHz Core(TM)2 Quad CPU and runs Ubuntu 12.04 LTS system and Apache2.

Table 5 lists the results for both the client and the server. The first line represents the load and CPU utilization without *LinkScope* and we ensure that no other routine processes are executed on both machines during the measurement. We can see that even when there are 100 probing process with 10Hz measurement rates, the average loads and average CPU utilizations are still very low on both machines, especially for the web server.

## 5 Related work

Network anomaly detection can be roughly divided into two categories: performance related anomalies and security related anomalies [35]. The performance related anomalies include transient congestion, file sever failure, broadcast storms and so on, and security related network anomalies are often due to DDoS attacks [36–39] that flood the network to prevent legitimate users from accessing the services. *PachScope* employs various performance metrics to detect a new class of target link flooding attacks (LFA).

Anomaly detection attempts to find patterns in data, which do not conform to expected normal behavior [40]. However, LFA can evade such detection because an attacker instructs bots to generate legitimate traffic to congest target links and the attack traffic will never reach the victim's security detection system. Instead of passively inspecting traffic for discovering anomalies, *LinkScope* conducts noncooperative active measurement to cover as many paths as possible and captures the negative effect of LFA on performance metrics.

Although active network measurement has been employed to detect network faults and connectivity problems [41–48], they cannot be directly used to detect and locate LFA because of two major reasons. First, since LFA will cause temporal instead of persistent congestion, existing systems that assume persistent connection problems cannot be used [41–43]. Second, since LFA avoids causing BGP changes, previous techniques that rely on route changes cannot be employed [44–47]. Moreover, the majority of active network measurement systems require installing software on both ends of a network path, thus having limited scalability. To the best of our knowledge, *LinkScope* is the first system that can conduct both end-to-end and hop-by-hop noncooperative measurement, and takes into account the anomalies caused by LFA.

Router-based approaches have been proposed to defend against LFA and other smart DoS attacks [6–8, 49–52], their effectiveness may be limited because they cannot be widely deployed to the Internet immediately. By contrast, *LinkScope* can be easily deployed because it conducts noncooperative measurement that only requires installation at one end of a network path. Although *LinkScope* cannot defend against LFA, it can be used along with traffic engineering tools to mitigate the effect of LFA.

Existing network tomography techniques cannot be applied to locate the target link, because they have many impractical assumptions (e.g., multicast [53], source routing [54]). Although binary tomography may be used for identifying faulty network links [55], it just provides coarse information [56] and and is not suitable for locating the link targeted by LFA, because it adopts assumptions for network fault (e.g., only one highly congested link in one path [57], faulty links nearest to the source [58]). LFA can easily invalidate them. Moreover, the probers in network tomography create a measurement mesh network [59,60] whereas in our scenarios there is only one or a few probers that may not communicate with each other.

## 6 Conclusion

In this paper, we propose a novel system, *LinkScope*, to detect a new class of target link-flooding attacks (LFA) and locate the target link or area whenever possible. By exploiting the nature of LFA that causes severe congestion on links that are important to the guard area, *LinkScope* employs both the end-to-end and the hop-by-hop network measurement techniques to capture abrupt performance degradation due to LFA. Then, it correlates the measurement data and the traceroute data to infer the target link or area. After addressing a number of challenging issues, we have implemented *LinkScope* with 7174 lines of C codes and conduct extensive evaluation in a testbed and the Internet. The results show that *LinkScope* can quickly detect LFA with high accuracy and low false positive rate. In future work, we will conduct large-scale and continuous measurements to evaluate *LinkScope* and investigate the optimal deployment of *LinkScope*.

## 7 Acknowledgment

We thank the reviewers for their comments and suggestions and Paul Krizak, in particular, for shepherding our paper. This work is supported in part by the CCF-Tencent Open Research Fund, the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396,60903185), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province.

## References

[1] Incapsula, "2013-2014 DDoS threat landscape report," 2014.

[2] ARBOR, "DDoS and security reports," http://www.arbornetworks.com/asert/, 2014.

[3] M. Geva, A. Herzberg, and Y. Gev, "Bandwidth distributed denial of service: Attacks and defenses," *IEEE Security and Privacy*, Jan.-Feb. 2014.

[4] A. Studer and A. Perrig, "The coremelt attack," in *Proc. ESORICS*, 2009.

[5] M. Kang, S. Lee, and V. Gligor, "The crossfire attack," in *Proc. IEEE Symp. on Security and Privacy*, 2013.

[6] S. Lee, M. Kang, and V. Gligor, "Codef: collaborative defense against large-scale link-flooding attacks," in *Proc. ACM CoNEXT*, 2013.

[7] S. Lee and V. Gligor, "Floc: Dependable link access for legitimate traffic in flooding attacks," in *Proc. IEEE ICDCS*, 2010.

[8] A. Athreya, X. Wang, Y. Kim, Y. Tian, and P. Tague, "Resistance is not futile: Detecting ddos attacks without packet inspection," in *Proc. WISA*, 2013.

[9] M. Crovella and B. Krishnamurthy, *Internet Measurement: Infrastructure, Traffic and Applications*. Wiley, 2006.

[10] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker, "On routing asymmetry in the internet," in *Proc. IEEE GLOBECOM*, 2005.

[11] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with Paris traceroute," in *Proc. ACM IMC*, 2006.

[12] A. Khan, T. Kwon, H. Kim, and Y. Choi, "AS-level topology collection through looking glass servers," in *Proc. ACM IMC*, 2013.

[13] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *Proc. ACM SIGCOMM*, 2002.

[14] X. Luo and R. Chang, "On a new class of pulsing Denial-of-Service attacks and the defense," in *Proc. NDSS*, 2005.

[15] E. Chan, X. Luo, W. Li, W. Fok, and R. K. Chang, "Measurement of loss pairs in network paths," in *Proc. ACM IMC*, 2010.

[16] X. Luo, E. Chan, and R. Chang, "Design and implementation of TCP data probes for reliable and metric-rich network path monitoring," in *Proc. USENIX ATC*, 2009.

[17] E. Chan, A. Chen, X. Luo, R. Mok, W. Li, and R. Chang, "TRIO: Measuring asymmetric capacity with three minimum round-trip times," in *Proc. ACM CoNEXT*, 2011.

[18] R. Koodli and R. Ravikanth, "One-way loss pattern sample metrics," RFC 3357, Aug. 2002.

[19] J. Sommers, P. Barford, and W. Willinger, "Laboratory-based calibration of available bandwidth estimation tools," *Microprocess. Microsyst.*, vol. 31, no. 4, pp. 222–235, 2007.

[20] N. Hu, L. Li, Z. M. Mao, P. Steenkiste, and J. Wang, "Locating internet bottlenecks: Algorithms, measurements, and implications," in *Proc. ACM SIGCOMM*, 2004.

[21] M. Allman, V. Paxson, and E. Blanton, "Rfc5681: Tcp congestion control," 2009.

[22] B. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster Analysis*, 5th ed. Wiley, 2011.

[23] B. Brodsky and B. Darkhovsky, *Non-Parametric Statistical Diagnosis Problems and Methods*. Kluwer Academic Publishers, 2000.

[24] Z. Durumeric, E. Wustrow, and J. Halderman, "Zmap: Fast internet-wide scanning and its security applications," in *Proc. 22nd USENIX Security Symposium*, 2013, pp. 605–619.

[25] M. Tozal and K. Sarac, "Tracenet: An internet topology data collector," in *Proc. ACM IMC*, 2010.

[26] J. Li, T. Ehrenkranz, and P. Elliott, "Buddyguard: A buddy system for fast and reliable detection of ip prefix anomalies," in *Proc. IEEE ICNP*, 2012.

[27] E. Katz-Bassett, H. Madhyastha, V. Adhikari, C. Scott, J. Sherry, P. Wesep, A. Krishnamurthy, and T. Anderson, "Reverse traceroute," in *Proc. USENIX NSDI*, 2010.

[28] A. Khan, T. Kwon, H. Kim, and Y. Choi, "As-level topology collection through looking glass servers," in *Proc. ACM IMC*, 2013.

[29] J. Padhye and S. Floyd, "Identifying the TCP behavior of web servers," in *Proc. ACM SIGCOMM*, 2001.

[30] Y. Lin, R. Hwang, and F. Baker, *Computer Networks: An Open Source Approach*. McGraw-Hill, 2011.

[31] Netfilter, http://www.netfilter.org.

[32] Planetlab, https://www.planet-lab.org.

[33] A. Dainotti, A. Botta, and A. Pescapè, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, 2012.

[34] htop, http://hisham.hm/htop/.

[35] M. Thottan and C. Ji, "Anomaly detection in ip networks," *IEEE Trans. on Signal Processing*, vol. 51, no. 8, 2003.

[36] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the dos and ddos problems," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, 2007.

[37] G. Loukas and G. Öke, "Protection against denial of service attacks: a survey," *The Computer Journal*, vol. 53, no. 7, 2010.

[38] S. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, 2013.

[39] M. Bhuyan, H. Kashyap, D. Bhattacharyya, and J. Kalita, "Detecting distributed denial of service attacks: Methods, tools and future directions," *The Computer Journal*, Mar. 2013.

[40] M. Bhuyan, D. Bhattacharyya, and J. Kalita, "Network anomaly detection: Methods, systems and tools," *Communications Surveys & Tutorials*, 2013.

[41] L. Quan, J. Heidemann, and Y. Pradkin, "Trinocular: Understanding internet reliability through adaptive probing," in *Proc. ACM SIGCOMM*, 2013.

[42] Y. Zhang, Z. Mao, and M. Zhang, "Detecting traffic differentiation in backbone ISPs with NetPolice," in *Proc. ACM IMC*, 2009.

[43] E. Katz-Bassett, H. Madhyastha, J. John, A. Krishnamurthy, D. Wetherall, and T. Anderson, "Studying black holes in the internet with hubble," in *Proc. USENIX NSDI*, 2008.

[44] Y. Liu, X. Luo, R. Chang, and J. Su, "Characterizing Inter-Domain Rerouting by Betweenness Centrality after Disruptive Events," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 5, 2013.

[45] W. Fok, X. Luo, R. Mok, W. Li, Y. Liu, E. Chan, and R. Chang, "Monoscope: Automating network faults diagnosis based on active measurements," in *Proc. IFIP/IEEE IM*, 2013.

[46] E. Chan, X. Luo, W. Fok, W. Li, and R. Chang, "Non-cooperative diagnosis of submarine cable faults," in *Proc. PAM*, 2011.

[47] Y. Zhang, Z. Mao, and M. Zhang, "Effective diagnosis of routing disruptions from end systems," in *Proc. USENIX NSDI*, 2008.

[48] X. Luo, L. Xue, C. Shi, Y. Shao, C. Qian, and E. Chan, "On measuring one-way path metrics from a web server," in *Proc. IEEE ICNP*, 2014.

[49] A. Shevtekar and N. Ansari, "A router-based technique to mitigate reduction of quality (roq) attacks," *Computer Networks*, vol. 52, no. 5, 2008.

[50] C. Zhang, Z. Cai, W. Chen, X. Luo, and J. Yin, "Flow level detection and filtering of low-rate DDoS," *Computer Networks*, vol. 56, no. 15, 2012.

[51] C. Chang, S. Lee, B. Lin, and J. Wang, "The taming of the shrew: mitigating low-rate tcp-targeted attack," *IEEE Trans. On Network Service Management*, Mar. 2010.

[52] X. Luo and R. Chang, "Optimizing the pulsing denial-of-service attacks," in *Proc. IEEE DSN*, 2005.

[53] R. Castro, M. Coates, G. Liang, R. Nowak, and B. Yu, "Network tomography: recent developments," *Statistical Science*, vol. 19, no. 3, 2004.

[54] L. Ma, T. He, K. Leung, A. Swami, and D. Towsley, "Identifiability of link metrics based on end-to-end path measurements," in *Proc. ACM IMC*, 2013.

[55] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "NetDiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data," in *Proc. ACM CoNEXT*, 2007.

[56] S. Zarifzadeh, M. Gowdagere, and C. Dovrolis, “Range tomography: combining the practicality of boolean tomography with the resolution of analog tomography,” in *Proc. ACM IMC*, 2012.

[57] H. Nguyen and P. Thiran, “The boolean solution to the congested ip link location problem:theory and practice,” in *Proc. IEEE INFOCOM*, 2007.

[58] N. Duffield, P. Avenue, and F. Park, “Network tomography of binary network performance characteristics,” *IEEE Trans. Information Theory*, vol. 52, no. 12, 2006.

[59] Q. Zheng and G. Cao, “Minimizing probing cost and achieving identifiability in probe based network link monitoring,” *IEEE Trans. Computers*, vol. 62, no. 3, 2013.

[60] Y. Zhao, Y. Chen, and D. Bindel, “Towards unbiased end-to-end network diagnosis,” *IEEE/ACM Transaction on Networking*, vol. 17, no. 6, 2009.

# Automatic and Dynamic Configuration of Data Compression for Web Servers

Eyal Zohar[*]
*Yahoo! Labs*
*Haifa, Israel*
*eyalz@yahoo-inc.com*

Yuval Cassuto
*Technion - Israel Institute of Technology*
*Department of Electrical Engineering*
*Haifa, Israel*
*ycassuto@ee.technion.ac.il*

## Abstract

HTTP compression is an essential tool for web speed up and network cost reduction. Not surprisingly, it is used by over 95% of top websites, saving about 75% of webpage traffic.

The currently used compression format and tools were designed over 15 years ago, with static content in mind. Although the web has significantly evolved since and became highly dynamic, the compression solutions have not evolved accordingly. In the current most popular web-servers, compression effort is set as a global and static compression-level parameter. This parameter says little about the actual impact of compression on the resulting performance. Furthermore, the parameter does not take into account important dynamic factors at the server. As a result, web operators often have to blindly choose a compression level and hope for the best.

In this paper we present a novel elastic compression framework that automatically sets the compression level to reach a desired working point considering the instantaneous load on the web server and the content properties. We deploy a fully-working implementation of dynamic compression in a web server, and demonstrate its benefits with experiments showing improved performance and service capacity in a variety of scenarios. Additional insights on web compression are provided by a study of the top 500 websites with respect to their compression properties and current practices.

## 1 Introduction

Controlling the performance of a web service is a challenging feat. Site load changes frequently, influenced by variations of both access volumes and user behaviors. Specifically for compression, the load on the server also depends on the properties of the user-generated content (network utilization and compression effort strongly depends on how compressible the data is). To maintain good quality-of-experience, system administrators must monitor their services and adapt their configuration to changing conditions on a regular basis. As web-related technologies get complex, ensuring a healthy and robust web service requires significant expertise and constant attention.

The increasing complexity raises the popularity of automated solutions for web configuration management [7, 25]. Ideally, an automatic configuration management should let system administrators specify high-level desired behaviors, which will then be fulfilled by the system [32]. In this paper we add such automation functionality for an important server module: HTTP compression.

HTTP compression is a tool for a web-server to compress content before sending it to the client, thereby reducing the amount of data sent over the network. In addition to typical savings of 60%-85% in bandwidth cost, HTTP compression also improves the end-user experience by reducing the page-load latency [29]. For these reasons, HTTP compression is considered an essential tool in today's web [34, 5], supported by all web-servers and browsers, and used by over 95% of the leading websites.

HTTP compression was standardized over 15 years ago [9], and with static web pages in mind, i.e., suitable for "compress-once, distribute-many" situations. But the dynamic nature of Web 2.0 requires web-servers to compress various pages on-the-fly for each client request. Therefore, today's bandwidth benefits of HTTP compression come with a significant processing burden.

The current most popular web-servers [28] (Apache, nginx, IIS) have an easily-deployable support for compression. Due to the significant CPU consumption of compression, these servers provide a configurable compression effort parameter, which is set as a global and static value. The problem with this configurable parameter, besides its inflexibility, is that it says little about

[*]Also with Technion - Israel Institute of Technology.

the actual amount of CPU cycles required to compress the outstanding content requests. Furthermore, the parameter does not take into account important factors like the current load on the server, response size, and content compressibility. As a result, web operators often have to blindly choose a compression level and hope for the best, tending to choose a low-effort compression-level to avoid overloading or long latencies.

Given its importance, HTTP compression has motivated a number of prior studies, such as [30, 6, 15]. However, our work is unique in considering simultaneously all aspects of compressed-web delivery: CPU, network bandwidth, content properties and server architectures. This combined study, and the concrete software modules provided along with it, are crucial to address the complexity of today's web services [18].

In this paper we present a deployable elastic compression framework. This framework solves the site operator's compression dilemma by providing the following features: 1) setting the compression effort automatically, 2) adjusting the compression effort to meet the desired goals, such as compression latency and CPU consumption, and 3) responding to changing conditions and availability of resources in seconds. We emphasize that the thrust of this framework is not improved compression algorithms, but rather a new algorithmic wrapping layer for optimizing the utilization of existing compression algorithms.

For better understanding of the problem at hand, Section 2 briefly surveys the key challenges of dynamic HTTP compression in cloud platforms. Since compression performance strongly depends on the content data itself, in Section 3 we analyze HTTP content from the top global websites, illuminating their properties with respect to their compression size savings and required computational effort. Then we turn to describe our solution. First we present in Section 4 a fully-functional implementation along with its constituent algorithms. Then, using a real-life workload, in Section 5 we demonstrate how the implementation operates. Section 6 reviews background of HTTP compression and related work, and Section 7 concludes and discusses future work.

## 2 Challenges

This section sketches some of the challenges of data compression in the dynamic content era. These challenges set the ground for the study that follows in subsequent sections.

### 2.1 Static vs. Dynamic Compression

Static content relates to files that can be served directly from disk (images/videos/CSS/scripts etc.). Static compression pre-compresses such static files and saves the compressed forms on disk. When the static content is requested by a decompression-enabled client (almost every browser), the web server delivers the pre-compressed content without needing to compress the content upon the client's request [27]. This mechanism enables fast and cheap serving of content that changes infrequently.

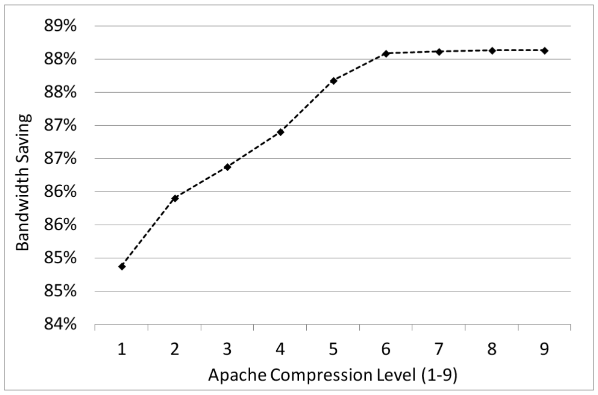
Dynamic content, in the context of this paper, relates to web pages that are a product of application frameworks, such as ASP.NET, PHP, JSP, etc. Dynamic web pages are the heart of the modern web [31]. Since dynamic pages can be different for each request, servers compress them in real time. As each response must be compressed on the fly, the dynamic compression is far more CPU intensive than static compression. Therefore, when a server is CPU bound it may be better not to compress dynamically and/or to lower the compression effort. On the other hand, at times when the application is bound by network or database capabilities, it may be a good idea to compress as much as possible.

### 2.2 CPU vs. Bandwidth Tradeoff

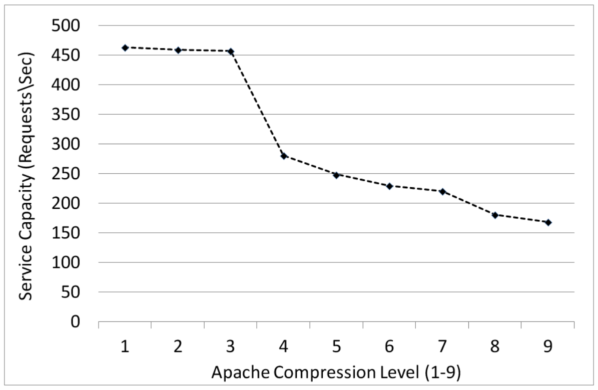
The focus in this paper is on compression of dynamic content, namely unique HTML objects generated upon client request. The uniqueness of these objects may be the result of one or more of several causes, such as personalization, localization, randomization and others. On the one hand, HTML compression is very rewarding in terms of bandwidth saving, typically reducing traffic by 60-85%. On the other hand, each response needs to be compressed on-the-fly before it is sent, consuming significant CPU time and memory resources on the server side.

Most server-side solutions allow choosing between several compression algorithms and/or effort levels. Generally speaking, algorithms and levels that compress better also run slower and consume more resources. For example, the popular Apache web-server offers 9 compression setups with generally increasing effort levels and decreasing output sizes.

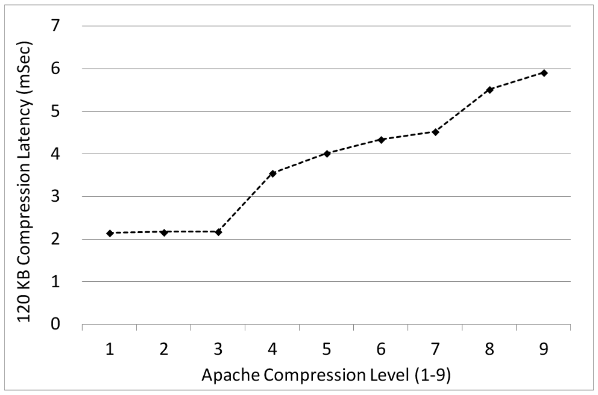
Figure 1a presents a typical bandwidth reduction achieved with all the 9 compression levels in an Apache site. We intentionally postpone the full setup details to Section 5, and just mention at this point that the average page size in this example is 120 KB. The complementary Figure 1b shows the CPU vs. bandwidth trade-off; the higher compression efforts of the upper levels immediately translate to lower capacities of client requests.



**(a)** Gain - bandwidth saving improves in upper levels



**(b)** Pain - service capacity shrinks in upper levels



**(c)** Pain - compression latency grows in upper levels

**Figure 1:** Compression gain and pain per level, of dynamic pages with average size of 120 KB

## 2.3 Latency: First-Byte vs. Download-Time Tradeoff

Web testing tools often use the Time To First Byte (TTFB) measurement as an indication of the web server's efficiency and current load. The TTFB is the time from when a browser sends an HTTP request until it gets the first byte of the HTTP response. Some tools [12] practically grade the web-server "quality" according to the TTFB value.

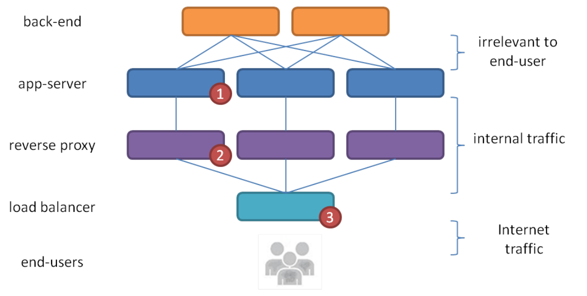


**Figure 2:** Compression location alternatives in the web server side.

When compression is in use by the server, the TTFB tends to get higher. This is because today's dynamic servers usually perform the following steps in a pure sequential manner: 1) page generation, 2) compression, and 3) transfer. Therefore, the larger the page and the higher the compression level, the larger the TTFB.

On the other hand, compression obviously reduces dramatically the complete download time of a page. Altogether, although the compression increases the TTFB, there is no doubt that this extra delay pays itself when considering the complete download time [14]. The open question is how high the compression level should be to reap download-time benefits without sacrificing latency performance. For example, Figure 1c shows the compression time of a 120 KB page, where the slowest level takes x3 more time than the fastest level, which is a typical scenario as we show in the sequel.

## 2.4 Where to Compress

Optimizing data compression in the web server is very promising, but simultaneously challenging due to the great richness and flexibility of web architectures. Even the basic question of *where* in the system compression should be performed does not have a universal answer fitting all scenarios. Compression can be performed in one of several different layers in the web server side. Figure 2 illustrates a typical architecture of a web-application server, where each layer may be a candidate to perform compression: 1) the application-server itself, 2) offloaded to a reverse-proxy, or 3) offloaded to a central load-balancer. On first glance all these options seem equivalent in performance and cost implications. However, additional considerations must be taken into account, such as a potential difficulty to replicate application-servers due to software licensing costs, and the risk of running CPU-intensive tasks on central entities like the load-balancer.

## 3 Web-Compression Study

In this work our focus is on compression of dynamic HTTP content in web-server environments. A study of compression has little value without examining and reasoning about the data incident upon the system. Therefore, in this section we detail a study we conducted on real-world HTTP content delivered by servers of popular web-sites. The results and conclusions of this study have shaped the design of our implementation and algorithms, and more importantly, they motivate and guide future work on algorithmic enhancements that can further improve performance. Another contribution of this study is the good view it provides on current compression practices, which reveals significant inefficiencies that can be solved by smarter compression.

The study examines HTML pages downloaded from top 500 global sites. The content of the pages is analyzed in many aspects related to their compression effort and size savings.

**Table 1:** Websites analysis - basic properties

| | |
|---|---|
| Sites supporting "gzip" format | 92% |
| Sites supporting "deflate" format | 9% |
| Average download size (compressed only) | 22,526 |
| Average uncompressed file | 101,786 |
| Compression ratio (best-median-worst) | 11%-25%-53% |
| Fully dynamic sites | 66% |

**Table 2:** Websites analysis - web-server survey.

| Developer | Share |
|---|---|
| Apache | 40.3% |
| nginx | 23.9% |
| gws | 15.3% |
| Microsoft | 6.5% |
| lighttpd | 1.7% |
| YTS | 0.9% |
| PWS | 1.1% |
| Others | 10.2% |

### 3.1 Setup

We fetched the list of the top 500 global sites from Alexa [2] in October 2012. For each site, we downloaded its main page at least once every hour with a gzip-enabled browser and saved it for further processing. Then, we emulated possible compression operations performed by the origin servers, by compressing the decompressed form of the pages using various tools and parameters. The analysis presented here is based on 190 consecutive downloads from each site in a span of one week.

### 3.2 Content Analysis

The first layer of the study is understanding the properties of popular content. Here we are interested in their compression ratios, their dynamism, and statistics on how website operators choose to compress them. We ran a basic analysis of 465 sites out of the top 500 sites. The rest are sites that return content that is too small to compress. A summary is presented in Table 1, where the numbers aggregate the entire set of one week snapshots from all the sites.

We start with examining what the supported compression formats are, by trying each of the formats maintained by IANA[21]. We find that the vast majority of the sites (85%) support "gzip" only, while 9% of that group support "gzip" and "deflate" only. (In the current context "gzip" and "deflate" in quotation marks refer to standard format names. The same terms are also used in other contexts to describe compression implementations, as explained in Section 6.) This means that our choice of gzip and deflate for our implementation and experiments is applicable to the vast majority of real-world HTTP content. As far as compression-ratio statistics go, the median compression ratio (across sites) is 25% (4:1 compression). The best compression ratio we measured is 11% (9:1) and the worst is 53% (∼2:1).

The next measurement of interest is the dynamism of web content, which is the variations of supplied data in time and across requesting clients. We found that 66% of the sites generated a unique page every time we have downloaded a snapshot. This is not surprising considering that dynamic pages are at the heart of Web 2.0. While this sample does not give an accurate prediction, it does attest to the general need for on-the-fly compression pursued in this paper.

An important statistic, especially for the choice of an implementation environment for our algorithms, relates to the server types used by popular sites. The most popular web-server turns out to be Apache, as illustrated in Table 2. These findings match an elaborate survey conducted in November 2012 [28]. This finding was the motivator to choose Apache as the platform for implementation and evaluation of our algorithms.

### 3.3 Performance Analysis

Beyond the analysis of the compression ratios and existing compression practices, it is useful for our framework to study how compression performance depends on the actual content. The results of this study hold the potential to guide server operators toward adopting compression practices with good balancing of effort and size savings. For that study we used our utilities to run performance

**Table 3:** Websites performance analysis.

| | |
|---|---|
| Sites that probably use zlib | 51% |
| Average zlib compression level | 5.02 |
| Average added traffic if used fastest level | +9.93% |
| Average reduced traffic if used best level | -4.51% |
| Sites using gzip's default (average 5.9-6.1) | 200 (47%) |

tests on the pages we had downloaded. These tools are available too for download from the project's site [36].

A summary is presented in Table 3, and later expanded in subsequent sub-sections. From the table it is learned that the zlib library emerges as a popular compression library in use today. We reached this conclusion by generating many different compressed forms of each downloaded page using different applications, and comparing the locally generated forms to the one we had downloaded. When a match is found, we project that the server is using the matching compression code and the parameters we use locally. Indeed, these tests show that at least 51% of the sites use zlib somewhere in their system: in the server, reverse-proxy, load-balancer, or at an external offload box. That is another reason, in addition to the popularity of Apache, for using Apache and zlib for the web-compression analysis below.

### 3.3.1 Compression level

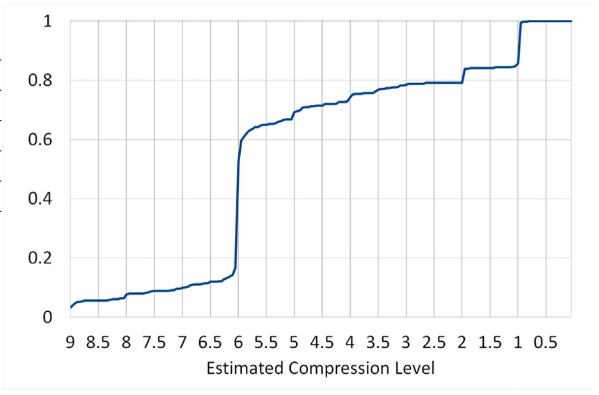
In this sub-section we estimate which compression level is used by each site. The results are presented in Figure 3a as a CDF curve. We found that the average estimated compression level in use is 5.02 and the median is 6, which is also zlib's default level.

### 3.3.2 Compression effort

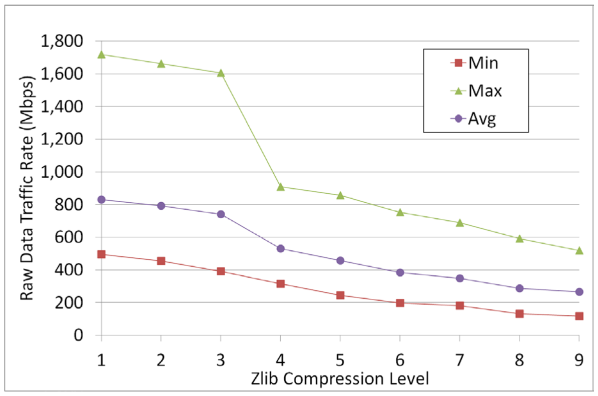
We compressed the contents from all the sites using all 9 levels, and examined the CPU effort induced by the compression levels. The results in Figure 3b show significant effort variation across sites per level. For example, in level 1, the slowest site (the one that required the maximal effort) required x3.5 more CPU power than the fastest site at the same level. In addition, there is at least one case in which level 1 in one site runs slower than level 9 for another site. Hence we conclude that the algorithmic effort, exposed to the user in the form of compression levels, cannot be used as a prediction for the CPU effort.

### 3.3.3 Fastest vs. slowest levels

Levels 1 and 9 are the extreme end points of the compression capabilities offered by zlib. As such, it is interesting to study the full tradeoff window they span between the fastest compression and the best-ratio one. Specifically, how much more effort is needed to move from level 1 to level 9, and what the gain is. The answer to this question is presented for all the sites in Figure 4a. The results show, again, that effort cannot be predicted based upon content size and compression level alone: effort can grow by x2 to x6.5 (*y* locations of the points), while the gain in traffic reduction is anything between 9% to 27% (*x* locations of the points).



**(a)** Compression levels used in practice, assuming that the sites use zlib-based compression code

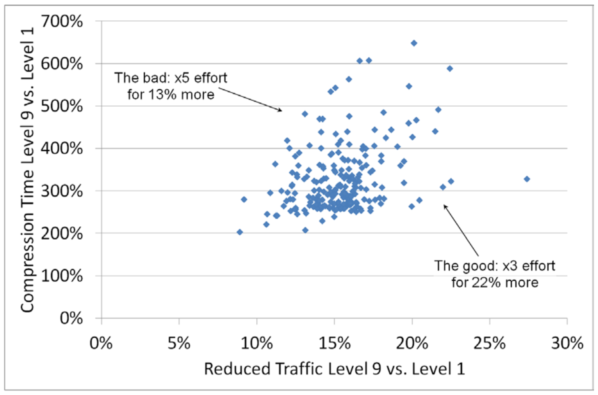


**(b)** CPU effort induced by each compression level (min, average, and max)

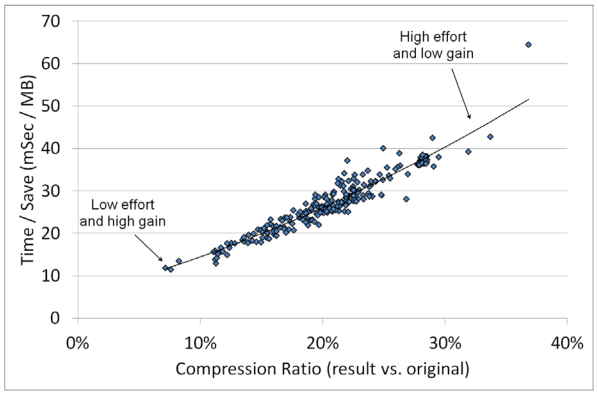
**Figure 3:** Top sites analysis - levels and CPU effort.

### 3.3.4 Compression speed vs. ratio

Another important finding for our framework is that the amount of redundancy in the page is highly correlated with low-effort compression, even at the upper levels. When the redundancy is high, zlib is able to find and eliminate it with little effort, thus reducing file sizes at low processing costs. These relations are depicted in Figure 4b, which presents the compression speed (high speed = low effort) versus the compression ratio (low ratio = strong compression) of all the sites we examined, when compressed locally with the default level 6.



**(a)** Comparing the costs and gains in moving from level 1 to level 9



**(b)** Time/save ratio vs. page compressibility, if all sites were using zlib level 6 (default)

**Figure 4:** Top sites analysis - costs, gains, and compressibility.

## 3.4 Considering the Cloud Pricing Plan

Continuing the study in the direction of web service over the cloud, we are now interested to find for each compression level the totality of the operational costs when deployed over a cloud service, i.e., the *combined* cost of computing and network bandwidth. For this part of the study we assumed an Amazon EC2 deployment, with the prices that were available at the time of the experiment. Clearly the results depend on the instantaneous pricing, and as such may vary considerably. Thus the results and conclusions of this study should be taken as an illustrative example. We further assume that the offered cloud services are fully scalable, stretching to unlimited demand with a linear increase in operational costs. Figure 5 shows the optimal level on a per-site basis, revealing that level 7 is the optimal level for most sites, but also showing that some sites gain more from level 6 or 8.

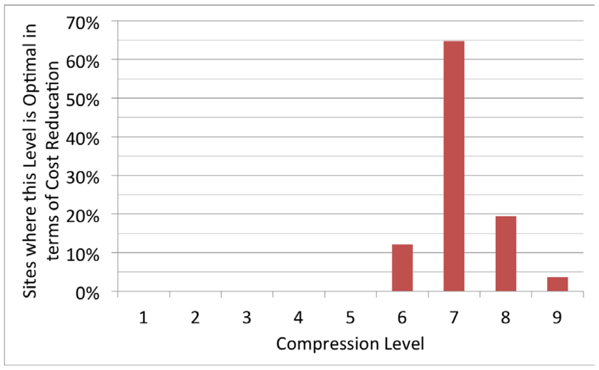


**Figure 5:** Top sites analysis - considering a momentary pricing plan of Amazon EC2. Cost reduction is relative to level 1 compression. The optimal level, presented as percentage of sites where each level is the optimal

## 4 Implementation

Our main contribution to compression automation is software/infrastructure implementations that endow existing web servers with the capability to monitor and control the compression effort and utility. The main idea of the implemented compression-optimization framework is to adapt the compression effort to quality-parameters and the instantaneous load at the server. This adaptation is carried out fast enough to accommodate rapid changes in demand, occurring in short time scales of seconds.

In this section we provide the details of two alternative implementations and discuss their properties: Both alternatives are found in the project's site [36] and are offered for free use and modification:

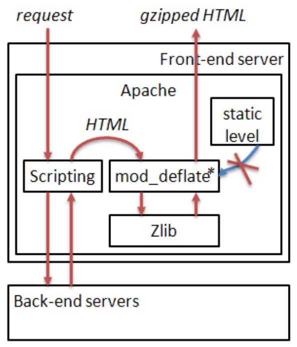
**System-A** - A modification of the deflate module of Apache (mod_deflate) that adapts the compression level to the instantaneous CPU load at the server.

**System-B** - Two separate modules work in parallel to offer a flexible but more complex solution: a monitor-and-set stand-alone process and an enhancement plugin for the compression entity.

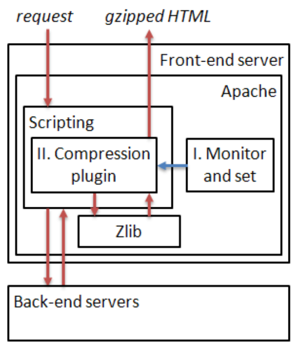
### 4.1 System-A

System-A is a transparent solution designed for seamless deployment and use in any working Linux-Apache environment. It does not require any changes in the working code and site structure, but requires a build of Apache with a modified *mod_deflate.c*. The modified module contains 100+ new lines of code in C language.

Figure 6a illustrates the system's architecture, which is very similar to a standard Apache. The only difference between a standard Apache and System-A, is that the *mod_deflate* module does not use the static compression level from the configuration file. Instead, the mod-



**(a)** System-A: Apache with a modified mod_deflate module



**(b)** System-B: monitor-and-set and a plugin

**Figure 6:** Implementation architectures

ule performs the following: 1) continuously checks the system's load, 2) remembers what was the last compression level in use, and 3) updates periodically the compression level, if needed.

The effort adaptation is performed in one-step increments and decrements. Proc. 1 gives a short pseudo-code that presents the basic idea in a straight-forward manner.

**Proc. 1** Simplified pseudo-code of the level modification phase in *mod_deflate.c*

```
1. if $next\_update > cur\_time$ then
2.   if $cpu > high\_threshold$ and $cur\_level > 1$ then
3.     $cur\_level \leftarrow cur\_level - 1$
4.   else if $cpu < low\_threshold$ and $cur\_level < 9$ then
5.     $cur\_level \leftarrow cur\_level + 1$
6.   end if
7.   $next\_update \leftarrow cur\_time + update\_interval$
8. end if
```

## 4.2 System-B

System-B takes a different approach – it is agnostic to the OS and web-server type, but it requires a change in the original site's code. Two separate modules in our implementation work in parallel to offer fast-adapting compression configuration. These are:

**Monitor and set** – Stand-alone process that monitors the instantaneous load incident on the machine, and chooses a compression setup to match the current machine conditions.

**Compression plugin** – Enhancement plugin for the compression entity that uses the chosen setup to compress the content. It operates in a fine granularity allowing to mix different setups among the outstanding requests.

Figure 6b illustrates the implementation structure. This solution compresses the HTML before it is being handed back from scripting to Apache. This allows the adaptive solution to bypass the static built-in compression mechanism and add the desired flexibility.

### 4.2.1 Module I – Monitor and Set

The monitor-and-set module runs as a background process on the same machine where compression is performed. Its function is essentially a control loop of the CPU load consisting of measuring the load and controlling it by setting the compression level. Its implementation as a separate process from the compression module is designed to ensure its operation even at high loads by proper prioritization.

### 4.2.2 Module II – Compression Plugin

The compression plugin is designed as an enhancement, rather than replacement, of an existing compression tool. This design allows our scheme to work with any compression tool chosen by the web-server operator or target platform. The sole assumption made about the compression tool in use is that it has multiple compression setups, ordered in non-descending effort levels. For example, the widely used zlib [13] compression tool offers a sequence of 9 setups with generally increasing effort levels and non-increasing compressed sizes.

This plugin code should be added at the end of an existing script code (like PHP), when the content is ready for final processing by the web-server. The plugin uses the setup provided by the monitor-and-set process and invokes the platform's compression tool with the designated setup. An important novelty of this algorithm is its ability to implement a non-integer setup number by mixing two setups in parallel. The fractional part of the setup number determines the proportion of requests to compress in each of the integer setups. For example, when the input setup number is 6.2, then 80% of the requests can be compressed with setup 6, while the rest are compressed with setup 7.

### 4.3 Practical Considerations

We now turn to discuss the details pertaining to the specific implementations we use in the evaluation. Part of the discussion will include special considerations when operating in a cloud environment. Fully functional versions of this code, for different environments, can be obtained from the project's web page [36].

CPU monitoring uses platform-dependent system tools, and further needs to take into account the underlying virtual environment, like the Amazon EC2 we use in the paper. Specifically, we need to make sure that the CPU load readout is an accurate estimate [19] of the processing power available for better compression in the virtual server. From experimentation within the Amazon EC2 environment, we conclude that Linux utilities give accurate CPU utilization readouts up to a cap of CPU utilization budget determined by the purchased instance size. For example, in an m1.small instance the maximal available CPU is about 40%, while m1.medium provides up to 80% CPU utilization. These CPU budgets are taken into account in the CPU threshold parameters.

When deploying in a real system, the compression level should not be adapted too frequently. To ensure a graceful variation in CPU load, it is advisable to choose a minimal interval of at least 0.5 second.

## 5 Proof of Concept Scenarios

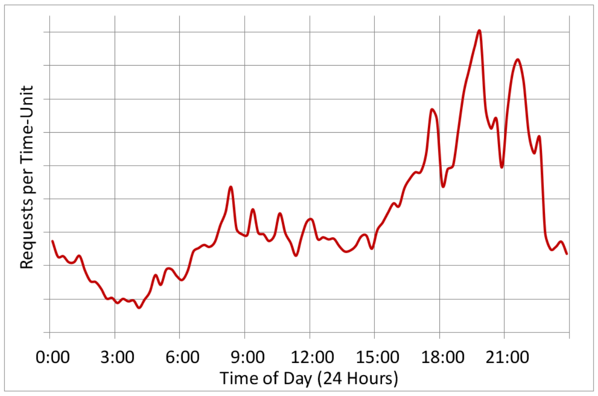
After detailing our implementation of elastic web compression, in this section we turn to report on usage scenarios and our experience with the code. For the study we choose the Amazon EC2 environment, for both its popularity and flexibility. In the study we compare, under different scenarios of interest, the performance of our implementation of elastic compression to the static compression currently employed in popular commercial web-servers.

Our implementation is next shown to offer the following features:

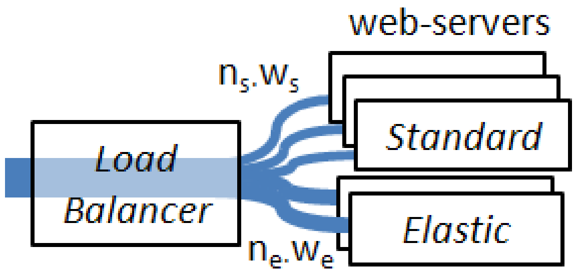
1. Protecting against sudden demand peaks and denial-of-service (DoS) attacks.

2. Trading server's free CPU time for bandwidth savings at low/medium loads.

Before dispatching to these individual scenarios, we describe the general setup of our experiments.



**(a)** Workload



**(b)** System

**Figure 7:** Experiment setup - the workload is divided between a collection of standard web-servers and servers that run our elastic compression.

### 5.1 General Setup

The workload and the general setup are illustrated in Figure 7. We use EC2 instances of type m1.small, each providing 1.7 GB memory and 1 EC2 Compute Unit [3]. Each instance is a single front-end web-server, with Ubuntu Server 12.04.2 LTS 64-bit, Apache 2.2.24 (Amazon) and PHP 5.3.23. For a side-by-side comparison with standard web-servers ("Standard"), we equipped one or more of the servers with our elastic implementation ("Elastic"). Monitoring of the instances' resources and billing status is performed with Amazon CloudWatch.

The workload is a 24-hour recording of 27,000 distinct users who visited a specific social network site. This workload is typical to sites of its kind, demonstrating low demand at 5:00 AM and high peaks around 8:00 PM, as illustrated in Figure 7a. We replay this workload by running multiple clients from multiple machines in AWS. Each front-end web-server receives a fraction of the requests in any given time, through a load-balancer. The fractions are depicted in Figure 7b as $w_s$ for each "Standard" server, and $w_e$ for each "Elastic" server. More details are provided in each scenario separately.

## 5.2 Case 1: Spike/DoS Protection

In this sub-section we use the elastic compression as a protection utility against sudden traffic peaks and/or organized DoS attacks.

The "Elastic" maximal deflate level is set to 6, which is the default and the most popular compression setup of Apache (gzip level 6), as we show in Section 3. All the web-servers share the overall traffic evenly, while one web-server is equipped with the elastic compression. To emulate a sudden peak or attack, we run the workload almost normally, with one exception: on 19:40-19:45 the service experiences a sudden 5 minutes attack, equivalent to 69% additional requests comparing with a normal high time-of-day. Figure 8a gives a zoom-in to the access pattern of a 1 hour interval containing the 5-minutes attack.

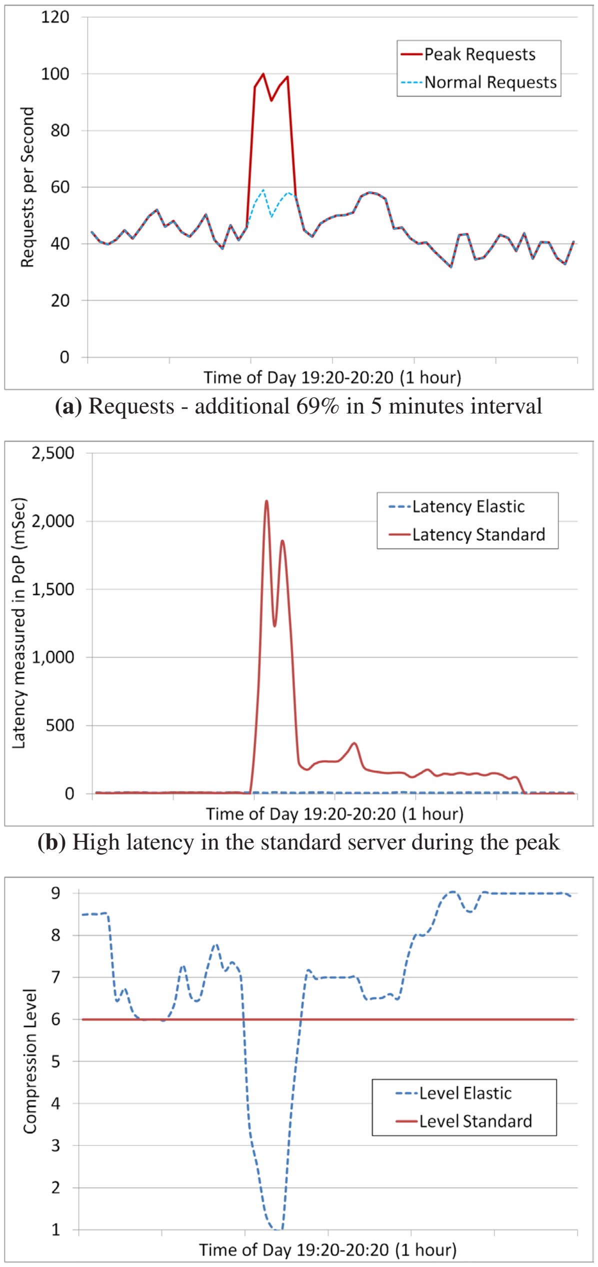
Figure 8b illustrates the projected latency at the clients who managed to complete a request, showing that the static server is saturated during the attack and also long after it. In addition (not shown in graph), almost half the client requests timed-out during the 5-minutes peak. Practically, all the "Standard" servers were out of order for at least 10 minutes. In reality, these attacks usually end up worse than that, due to client retransmission and/or servers that fail to recover. Figure 8c shows how the elastic compression handles the attack: it lowers the compression effort to minimum during the attack, until it feels that the server is no longer in distress.

## 5.3 Case 2: Compress More

We consider a standard web-server running low-effort compression. More specifically, the baseline is "Standard" Apache servers running compression level 1 (a.k.a fastest).

In this experiment, all the servers receive an equal share of the traffic throughout the day – meaning $w_s = w_e$. While a "Standard" server uses the same compression setup all day long, an "Elastic" server selectively changes the compression level: from 1 (fastest) to 9 (slowest). It changes the compression level according to the sensed load at the server, in order to compress more than the standard server when the end-users' demand is relatively low.

The standard machine's CPU follows the end-users' request pattern tightly, leaving large portion of the CPU unused most of the day. The elastic solution uses the free CPU for higher-effort compression most of the day, saving additional bandwidth, whenever there are enough unused resources for it. When demand is high, the elastic solution returns to low-effort compression, staying below the maximal allowed CPU consumption. In the given scenario, the adaptive compression level managed to save 10.62% of the total traffic volume during the 24-hours experiment.



**(a)** Requests - additional 69% in 5 minutes interval

**(b)** High latency in the standard server during the peak

**(c)** Elastic compression level, as used in practice

**Figure 8:** Case 3: handling a sudden peak and/or DoS attack.

## 5.4 Case 3: Reducing the 95th Percentile Latency

We now evaluate elastic compression on the real-world workload of another big web infrastructure[1]. While serv-

[1]The source has asked to remain anonymous, keeping its systems' structure and performance hidden from hostiles

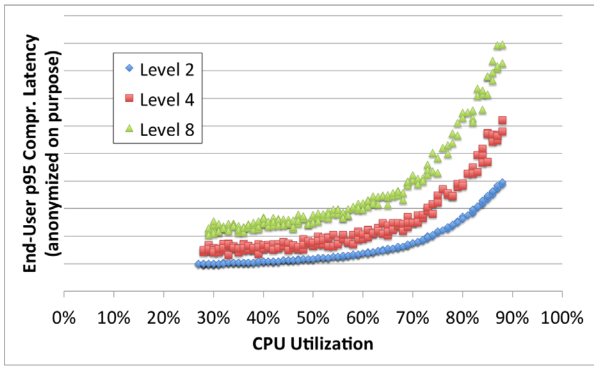


**Figure 9:** Reducing p95 latency: normalized compression times at different levels as a function of CPU utilization.

ing millions requests a second, its web servers strive to reduce response generation times for providing optimal user experience. Therefore, a compression strategy in such an environment should be as lightweight as possible while trying to minimize the needed bandwidth.

Measuring the latency of active real-world services is tricky, because the service provider is mostly worried about loosing customers that experience the worst latencies. Hence, it is a common practice to consider percentiles and not just the average. As compression times vary heavily among different responses we consider percentile p95 a representative of "heavy compressions".

In the following experiment we compare compression times and response sizes of levels 2, 4 and 8 as a function of CPU utilization at a web server. Figure 9 demonstrates that the differences between compression times at different levels grow super linearly as a function of CPU utilization[2]. This presents an opportunity to sacrifice 10% of the egress bandwidth under heavy load in order to drastically (x2) reduce the compression time.

## 6 Background and Related Work

Data compression and its effect on computer systems is an important and well studied problem in the literature. Here we survey some background on compression tools, and related work on compression-related studies.

It is important to first note that the popular compression term gzip stands for several different things: a software application [16], a file format [8, 9], and an HTTP compression scheme. Gzip uses the *deflate* compression algorithm, which is a combination of the LZ77 algorithm [35] and Huffman coding. Gzip was officially adopted by the web community in the HTTP/1.1 specifications [11]. The standard allows a browser to declare its gzip-decompression capability by sending the server an HTTP request field in the form of "Accept-Encoding: gzip". Gzip is the most broadly supported compression method as of today, both by browsers and by servers. The server, if supports gzip for the requested file, sends a compressed version of the file, prefixed by an HTTP response header that indicates that the returned file is compressed. All popular servers have built-in support or external modules for gzip compression. For example, Apache offers mod_deflate (in gzip format, despite the misleading name) and Microsoft IIS and nginx have built-in support.

It became a standard in the major compression tools to offer an effort-adjustment parameter that allows the user to trade CPU for bandwidth savings. In gzip the main parameter is called *compression level*, which is a number in the range of 1 ("fastest") to 9 ("slowest"). Lower compression levels result in a faster operation, but compromise for size. Higher levels result in a better compression, but slower operation. The default level provides an accepted compromise between compression ratio and speed, and is equivalent to compression level 6.

Industrial solutions for gzip include PCI-family boards [22, 1], and web accelerator boxes [10] that offload the CPU-intensive compression from the servers. Although these solutions work in relatively high speeds, they induce additional costs on the website owner. These costs make the hardware solutions inadequate for websites that choose compression for cost reduction in the first place.

There is an extensive research on compression performance in the context of energy-awareness in both wireless [4] and server [23] environments. Inline compression decision [6] presents an energy-aware algorithm for Hadoop that answers the "to compress or to not compress" question per MapReduce job. Similarly, fast filtering for storage systems [17] quickly evaluates the compressibility of real-time data, before writing it to storage. Fine-grain adaptive compression [30] mixes compressed and uncompressed packets in attempt to optimize throughput when the CPU is optionally the bottleneck in the system. A more recent paper [15] extends the mixing idea to scale down the degree of compression of a single document, using a novel implementation of a parallelized compression tool. A cloud related adaptive compression for non-web traffic [20] focuses on how to deal with system-metric inaccuracy in virtualized environments. We found that this inaccuracy problem, reported in July 2011 [19], no longer exists in Amazon EC2 today. To the best of our knowledge, our paper presents the first adaptive web compression that takes a complex cost-aware decision with multiple documents.

In this paper we use the gzip format and zlib [13] software library to demonstrate our algorithms, because

[2]The Y-axis values were normalized/anonymized on purpose, per the source's request.

both are considered the standard for HTTP compression. Nevertheless, the presented algorithms are not limited to any particular compression technology or algorithm; they can encompass many different compression algorithms and formats. In particular, they can also benefit from inline compression techniques [24, 33, 26] when support for these formats is added to web servers and browsers.

## 7 Conclusions

In this paper we had laid out a working framework for automatic web compression configuration. The benefits of this framework were demonstrated in several important scenarios over real-world environments. We believe that this initial work opens a wide space for future research on compression cost optimization in various platforms, including cloud-based services. Building on the main functionality of our proposed implementation, future implementations and algorithms can improve cost by tailoring compression to more system architecture and content characteristics.

## Acknowledgements

We would like to thank Alexander Tzigirintzev and Yevgeniy Sabin for their implementation of System-A project presented above, and Roy Mitrany for his invaluable feedback and insight in this project.

The authors would like to thank the anonymous LISA 2014 reviewers and our shepherd, Matt Simmons, for their comments and suggestions that considerably helped us to improve the final version.

This work was supported in part by a European Union CIG grant, and by the Israeli Ministry of Science and Technology.

## Availability

Our code and projects mentioned in this paper, are free software, available on a web site, along with additional information and implementation details, at

`http://eyalzo.com/ecomp`

## References

[1] AHA GZIP Compression/Decompression Accelerator. `http://www.aha.com/index.php/products-2/data-compression/gzip/`.

[2] Alexa Internat, Top Sites. 1996. `http://www.alexa.com/topsites`.

[3] Amazon EC2 Instance Types. April 2013. `http://aws.amazon.com/ec2/instance-types/`.

[4] K. Barr and K. Asanović. Energy Aware Lossless Data Compression. *Proc. of MobiSys*, 2003.

[5] A. Bremler-Barr, S. T. David, D. Hay, and Y. Koral. Decompression-Free Inspection: DPI for Shared Dictionary Compression over HTTP. In *Proc. of INFOCOM*, 2012.

[6] Y. Chen, A. Ganapathi, and R. H. Katz. To Compress or Not to Compress - Compute vs. IO Tradeoffs for MapReduce Energy Efficiency. In *Proc. of SIGCOMM Workshop on Green Networking*, 2010.

[7] Y. Chess, J. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[8] P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3. *RFC 1951*, May 1996.

[9] P. Deutsch. GZIP File Format Specification Version 4.3. *RFC 1952*, May 1996.

[10] F5 WebAccelerator. `http://www.f5.com/products/big-ip/big-ip-webaccelerator/`.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. *RFC 2616*, June 1999.

[12] W. Foundation. Web page performance test. 2014. `http://www.webpagetest.org`.

[13] J. L. Gailly and M. Adler. Zlib Library. 1995. `http://www.zlib.net/`.

[14] J. Graham-Cumming. Stop worrying about time to first byte (ttfb). 2012. `http://blog.cloudflare.com/ttfb-time-to-first-byte-considered-meaningles`.

[15] M. Gray, P. Peterson, and P. Reiher. Scaling Down Off-the-Shelf Data Compression: Backwards-Compatible Fine-Grain Mixing. In *Proc. of ICDCS*, 2012.

[16] Gzip Utility. 1992. `http://www.gzip.org/`.

[17] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *Proc. of FAST*, 2013.

[18] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proc. of IMC*, 2013.

[19] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke. Evaluating Adaptive Compression to Mitigate the Effects of Shared I/O in Clouds. In *Proc. of IPDPS Workshop*, 2011.

[20] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke. Adaptive Online Compression in Clouds - Making Informed Decisions in Virtual Machine Environments. *Springer Journal of Grid Computing*, 2013.

[21] IANA - HTTP parameters. May 2013. `http://www.iana.org/assignments/http-parameters/`.

[22] Indra Networks PCI Boards. `http://www.indranetworks.com/products.html`.

[23] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proc. of SYSTOR*, 2009.

[24] R. Lenhardt and J. Alakuijala. Gipfeli-High Speed Compression Algorithm. In *Proc. of Data Compression Conference (DCC)*, 2012.

[25] Z. Li, D. Levy, S. Chen, and J. Zic. Explicitly controlling the fair service for busy web servers. In *Proc. of ASWEC*, 2007.

[26] LZO Data Compression Library. `http://www.oberhumer.com/opensource/lzo/`.

[27] Microsoft IIS - Dynamic Caching and Compression. `http://www.iis.net/overview/reliability/dynamiccachingandcompression`.

[28] Netcraft Web Server Survey. November 2012. `http://news.netcraft.com/archives/2012/11/01/november-2012-web-server-survey.html`.

[29] S. Pierzchala. Compressing Web Content with mod_gzip and mod_deflate. In *LINUX Journal*, April 2004. `http://www.linuxjournal.com/article/6802`.

[30] C. Pu and L. Singaravelu. Fine-Grain Adaptive Compression in Dynamically Variable Networks. In *Proc. of ICDCS*, 2005.

[31] A. Ranjan, R. Kumar, and J. Dhar. A Comparative Study between Dynamic Web Scripting Languages. In *Proc. of ICDEM*, pages 288–295, 2012.

[32] S. Schwartzberg and A. L. Couch. Experience in Implementing an HTTP Service Closure. In *LISA*, 2004.

[33] snappy Project - A Fast Compressor/Decompressor. `http://code.google.com/p/snappy/`.

[34] S. Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, Dec. 2008.

[35] J. Ziv and A. Lempel. A Universal Algorithm for Sequential data Compression. *Information Theory, IEEE Transactions on*, 1977.

[36] E. Zohar and Y. Cassuto. Elastic compression project homepage. 2013. `http://www.eyalzo.com/projects/ecomp/`.

# The Truth About MapReduce Performance on SSDs

*Karthik Kambatla*[†‡], *Yanpei Chen*[†]
*{kasha, yanpei}@cloudera.com*
[†]*Cloudera Inc.*
[‡]*Dept. of Computer Science, Purdue University.*

## Abstract

Solid-state drives (SSDs) are increasingly being considered as a viable alternative to rotational hard-disk drives (HDDs). In this paper, we investigate if SSDs improve the performance of MapReduce workloads and evaluate the economics of using PCIe SSDs either in place of or in addition to HDDs. Our contributions are (1) a method of benchmarking MapReduce performance on SSDs and HDDs under constant-bandwidth constraints, (2) identifying *cost-per-performance* as a more pertinent metric than *cost-per-capacity* when evaluating SSDs versus HDDs for performance, and (3) quantifying that SSDs can achieve up to 70% higher performance for 2.5x higher cost-per-performance.

**Keywords:.** MapReduce, Analytics, SSD, flash, performance, economics.

## 1 Introduction

Solid-state drives (SSDs) are increasingly used for a variety of performance-critical workloads, thanks to their low latency (lack of seek overheads) and high throughput (bytes-per-second and IOPS). The relatively high *cost-per-capacity* of SSDs has limited their use to smaller datasets until recently. Decreasing prices [16] and low power-consumption [13] make them a good candidate for workloads involving large volumes of data. The lack of seek overhead gives them a significant advantage over traditional hard disk drives (HDDs) for random-access workloads such as those in key-value stores.

In this paper, we investigate the economics of using SSDs to improve the performance of MapReduce [8], a widely-used big-data analytics platform. As MapReduce represents an important software platform in the datacenter, the performance tradeoffs between SSDs and HDDs for MapReduce offers critical insights for designing both future datacenter server architectures and future big-data application architectures.

MapReduce is traditionally considered to be a sequential access workload. A detailed examination of the MapReduce IO pipeline indicates that there are IO patterns that benefits from the hardware characteristics of SSDs. Past studies on MapReduce SSD performance have not yet accurately quantified any performance gains, mostly due to previous hardware limits constraining studies to be simulation based, on unrealistic virtualized environments, or comparing HDD and SSD setups of different bandwidths (Section 2).

Our MapReduce benchmarking method seeks to compare HDDs and PCIe SSDs under constant bandwidth constraints (Section 3). We selected our hardware to answer the following questions: (1) when setting up a new cluster, how do SSDs compare against HDDs of same aggregate bandwidth, and (2) when upgrading an HDDs-only cluster, should one add SSDs or HDDs for better performance. We measured performance for a collection of MapReduce jobs to cover several common IO and compute patterns.

Our results quantify the MapReduce performance advantages of SSDs and help us identify how to configure SSDs for high MapReduce performance (Section 4). Specifically, we find that

1. For a new cluster, SSDs deliver up to 70% higher MapReduce performance compared to HDDs of equal aggregate IO bandwidth.
2. Adding SSDs to an existing HDD cluster improves performance if configured properly. SSDs in hybrid SSD/HDD clusters should be divided into multiple HDFS and shuffle local directories.

Beyond the immediate HDD versus SSD tradeoffs, a broader implication of our study is that the choice of storage media should consider *cost-per-performance* in addition to the more common metric of *cost-per-capacity* (Section 5). As a key benefit of SSDs is performance, one can argue that cost-per-performance is the more important metric. Our results indicate that SSDs deliver 0-70% higher MapReduce performance, depending on workload. On average, this translates to 2.5x higher cost-per-performance, a gap far smaller than the orders-of-magnitude difference in cost-per-capacity.

## 2 Background and Related Work

### 2.1 SSDs vs HDDs

The biggest advantage of SSDs over HDDs is the high IOPS. SSDs achieve this by avoiding the physical disk rotation and seek time. The sequential IO bandwidth is also higher: by measuring the time taken to copy a large

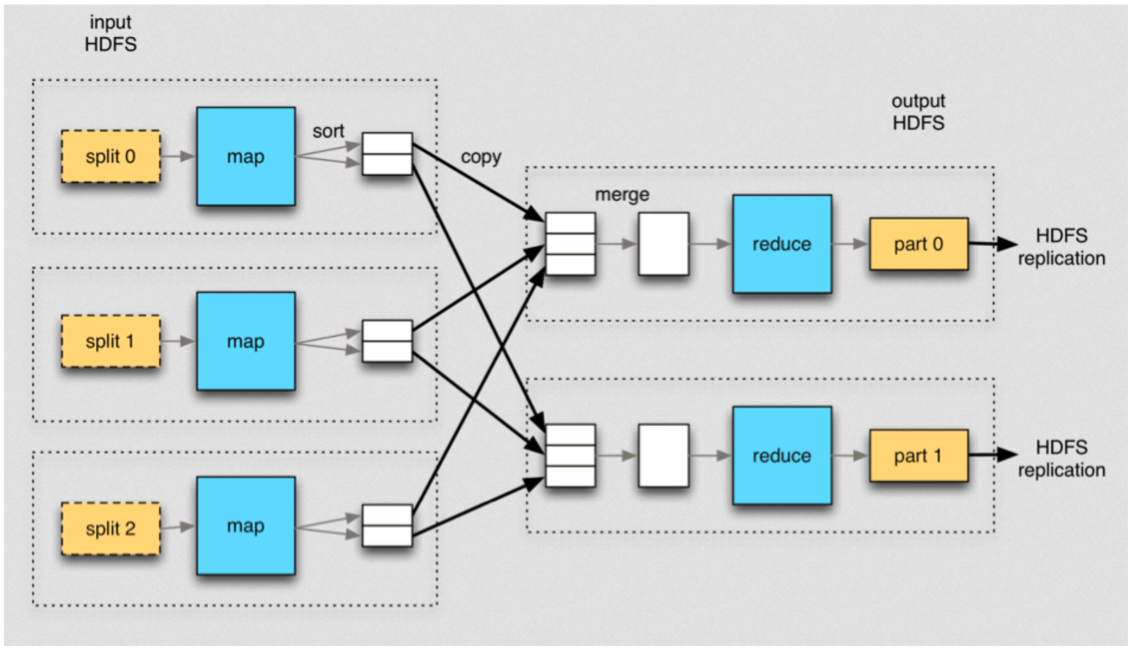


Figure 1: MapReduce Dataflow. Source: [19]

file, we found our HDDs could each support ∼120MBps of sequential read or write, while our SSDs were each capable of ∼1.3GBps sequential read or write.

The performance benefits of SSD compared to HDDs depend on the workload. For sequential I/O workloads, one can use multiple HDDs in parallel (assuming the application allows parallel access) to extract bandwidth comparable to SSD. For example, one can use 10 HDDs of 120MBps to match the 1GBps bandwidth of an SSD. On the other hand, realizing comparable bandwidth for random I/O workloads can be far more expensive, as one would need many more HDDs to offset the seek latency. For example, if an HDD delivers an effective bandwidth of 10 MBps for random accesses of a few MBs of data each IO, one needs to use at least 100 of them to achieve the same 1GBps aggregate bandwidth.

## 2.2 MapReduce – Dataflow

MapReduce [8] is a data-parallel processing framework designed to process large volumes of data in parallel on clusters of machines. In Apache Hadoop [1], a widely-used open-source MapReduce implementation, the execution is split into map, shuffle, reduce phases. Map and reduce phases are split into multiple tasks, each task potentially running on a different machine. Each map task takes in a set of key-value pairs ($< key, value >: list$), applies the map function to each pair and emits another set of key-value pairs ($< key, value >: list$). The shuffle phase partitions the map output from all map tasks such that all values corresponding to a key are in the same partition ($< key, value : list >: list$), each partition can be on a different node. Each reduce task picks these partitions, applies the reduce function per key, and writes the output to HDFS. Figure 1 captures the flow of data in a typical MapReduce job.

The effect of storage media, particularly SSD versus HDD, depends on the average read/write size and the randomness of data accesses. A typical MapReduce job exhibits two kinds of data accesses:

**Large, sequential HDFS accesses.** The job reads input splits from HDFS initially, and writes output partitions to HDFS at the end. Each task (dotted box) performs relatively long sequential IO of 100s of MBs. When multiple tasks are scheduled on the same machine, they can access the disks on the machine in parallel, with each task accessing its own input split or output partition. Thus, an HDD-only configuration of 11 disks of 120MBps each can potentially achieve HDFS read/write bandwidth comparable to a SSD drive of 1.3GBps.

**Small, random reads and writes of shuffle intermediate data.** MapReduce partitions each map output across all the reduce tasks. This leads to significantly lower IO size. For example, suppose a job has map tasks that each produces 1GB of output. When divided among, say, 1,000 reduce tasks, each reduce task fetches only 1MB. Analysis of our customer traces indicate that many deployments indeed have a per-reduce shuffle granularity of just a a few MBs or even lower.

Table 1: Storage configurations used

| **Setup** | **Storage** | **Capacity** | **Sequential R/W Bandwidth** | **Price (USD)** |
|---|---|---|---|---|
| **HDD-6** | 6 HDDs | 12 TB | 720 MBps | 2,400 |
| **HDD-11** | 11 HDDs | 22 TB | 1300 MBps | 4,400 |
| **SSD** | 1 SSD | 1.3 TB | 1300 MBps | 14,000 |
| **Hybrid** | 6 HDDs + 1 SSD | 13.3 TB | 2020 MBps | 16,400 |

The number of concurrent accesses on a node determines the extent of I/O multiplexing on the disks, which in turn depends on the stage of job execution. The number of map or reduce tasks per node determines the number of concurrent HDFS read/write accesses. During the shuffle phase, the map-side sort IO concurrency is determined by the total number of merge-sort threads used across all map tasks on a node. The network copy concurrency comes from the number of map-side threads serving the map outputs and the number of reduce-side threads remotely fetching map outputs. The reduce-side sort IO concurrency is also determined by the number of merge-sort threads on the node. In practice, independent of IO concurrency, there is negligible disk I/O for intermediate data that fits in memory, while a large amount of intermediate data leads to severe load on the disks.

A further dimension to consider is compression of HDFS and intermediate data. Compression is a common technique to shift load from IO to CPU. Map output compression is turned on by default in Cloudera's distribution including Apache Hadoop (CDH), as most common kinds of data (textual, structured numerical) are readily compressible. Job output compression is disabled by default in CDH for compatibility with historical versions. Tuning compression allows us to examine tradeoffs in storage media under two different IO and CPU mixes.

Based on the MapReduce dataflow and storage media characteristics, we hypothesize that:

1. SSDs improve performance of shuffle-heavy jobs.
2. SSDs and HDDs perform similarly for HDFS-read-heavy and HDFS-write-heavy jobs.
3. For hybrid clusters (both SSDs and HDDs), using SSDs for intermediate shuffle data leads to significant performance gains.
4. All else equal, enabling compression decreases performance differences by shifting IO load to CPU.

## 2.3 Prior work

A body of work on SSD performance for MapReduce and other big data systems is still emerging. To date, progress on this area has been limited by the cost and (un)availability of SSDs.

An early work on HDFS SSD performance used OS buffer cache to simulate a fast SSD [4]. The study focused on Apache HBase [2] performance, and found various long code paths in HDFS client affected read throughput and prevented the full potential of SSDs to be realized. Some of the bottlenecks have since been eliminated from HDFS.

Another study used real SSDs, but on a virtualized cluster, i.e., multiple virtualized Hadoop workers on a single physical machine [12]. The experiments found that Hadoop performs up to 3x better on SSDs. It remains unclear how the results translate to non-virtualized environments or environments where every virtualized node is located on a separate physical node.

A recent follow up to [4] simulated SSD performance as a tiered cache for HBase [9]. It found that under certain cost and workload models, a small SSD cache triples performance while increasing monetary cost by 5%.

The closest work to ours compared Hadoop performance on actual SSDs and HDDs [14], albeit on hardware with non-uniform bandwidth and cost. The study runs the *Terasort* benchmark on different storage configurations and found that SSD can accelerate the shuffle phase of the MapReduce pipeline, as we already hypothesized based on MapReduce IO characteristics.

Working with Cloudera's hardware partners, we designed our experiments to cover gaps in prior studies. We compare MapReduce performance on actual HDDs and SSDs, without virtualization, using storage hardware of comparable bandwidths, with MapReduce configurations optimized via experience at Cloudera's customers.

## 3 Experimental setup

Our choice of hardware and MapReduce benchmarks is guided by the following considerations:

- We compare SSDs vs HDDs performance under equal-bandwidth constraints. An alternative is to compare performance for equal-costs. We selected the equal-bandwidth setup because it reveals the intrinsic features of the technology without the impact of variable economic dynamics.
- We considered both SSDs/HDDs as storage for a new cluster, and SSDs/HDDs as additional storage

Table 2: Descriptions of MapReduce jobs

| Job | Description |
|---|---|
| Teragen | HDFS write job, with 3-fold replication that heavily uses the network. |
| Terasort | Job with 1:1:1 HDFS read, shuffle, and HDFS write. |
| Teravalidate | HDFS read-heavy job that also does sort order validation (mostly HDFS read), with some small IO in shuffle and HDFS write. |
| Wordcount | CPU-heavy job that heavily uses the map-side combiner. |
| Teraread | HDFS read-only job, like Teravalidate, except no sort order validation and no reduce tasks. |
| Shuffle | Shuffle-only job, modified from randomtextwriter in hadoop-examples. |
| HDFS Data Write | HDFS write-only job, like Teragen, except with 1-fold replication. |

Table 3: Data size and CPU utilization for the MapReduce jobs. Values are normalized against Terasort.

| Job | Input size | Shuffle size | Output size | CPU utilization |
|---|---|---|---|---|
| Teragen | 0 | 0 | 3 | 0.99 |
| Terasort | 1 | 1 | 1 | 1.00 |
| Teravalidate | 1 | 0 | 0 | 0.60 |
| Wordcount | 1 | 0.07 | 0.09 | 1.47 |
| Teraread | 1 | 0 | 0 | 1.23 |
| Shuffle | 0 | 1 | 0 | 1.01 |
| HDFS Data Write | 0 | 0 | 1 | 0.93 |

to enhance an existing HDD cluster. Both scenarios are relevant and of interest to enterprise customers.

- Our benchmark includes a series of MapReduce jobs to cover common IO and compute patterns seen in customer workloads. We deliberately deferred a more advanced method of measuring performance for multi-job workloads [5, 6]. The stand-alone, one-job-at-a-time method allows us to more closely examine MapReduce and storage media interactions without the impact of job scheduling and task placement algorithms.

## 3.1 Hardware

We used PCIe SSDs with 1.3TB capacity costing $14,000 each, and SATA HDDs with 2TB capacity costing $400 each. Each storage device is mounted with the Linux ext4 file system, with default options and 4KB block size. The machines are Intel Xeon 2-socket, 8-cores, 16-threads, with 10Gbps Ethernet and 48GB RAM. They are connected as a single rack cluster.

To get a sense of the user-visible storage bandwidth without HDFS and MapReduce, we measured the duration of copying a 100GB file to each storage device. This test indicates the SSDs can do roughly 1.3GBps sequential read and write, while the HDDs have roughly 120MBps sequential read and write.

Table 1 describes the storage configurations we evaluate. The SSD and HDD-11 setups allow us to compare SSDs vs HDDs on an equal-bandwidth basis. The HDD-6 setup serves as a baseline of IO-constrained cluster. The HDD-6, HDD-11, and Hybrid setups allow us to investigate the effects of adding either HDDs or SSDs to an existing cluster.

## 3.2 MapReduce jobs

Table 2 describes the MapReduce benchmark jobs that we use. Each is either a common benchmark, or a job constructed specifically to isolate a stage of the MapReduce IO pipeline.

More details on the jobs and our measurement method:

- Each job is set to shuffle, read, write, or sort 33GB of data per node.
- Where possible, each job runs with either a single wave of map tasks (Teragen, Shuffle, HDFS Data Write), or a single wave of reduce tasks (Terasort, Wordcount, Shuffle).
- We record average and standard deviation of job duration from five runs.
- We clear the OS buffer cache on all machines between each measurement.
- We used collectl to track IO size, counts, bytes, merges to each storage device, as well as network and CPU utilization.

Note that the jobs here are IO-heavy jobs selected and sized specifically to compare two different storage media. In general, real-world customer workloads have a variety of sizes and create load for multiple resources including IO, CPU, memory, and network.

Table 4: Performance-relevant MapReduce 2 configurations.

| **Configuration** | **Value** |
|---|---|
| mapreduce.task.io.sort.mb | 256 |
| mapreduce.task.io.sort.factor | 64 |
| mapreduce.task.io.sort.spill.percent | 0.8 |
| mapreduce.reduce.shuffle.parallelcopies | 10 |
| HDFS blocksize | 128 MB |
| yarn.nodemanager.resource.memory-mb | RAM size |
| yarn.nodemanager.resource.cpu-vcores | # of CPU threads |
| mapreduce.map.memory.mb | 1024 |
| mapreduce.reduce.memory.mb | 1024 |
| mapreduce.map.cpu.vcores | 1 |
| mapreduce.reduce.cpu.vcores | 1 |
| mapreduce.map.java.opts | -Xmx1000 |
| mapreduce.reduce.java.opts | -Xmx1000 |
| yarn.scheduler.minimum-allocation-mb | 256 |
| mapreduce.job.reduce.slowstart.completedmaps | 0.8 |
| mapreduce.map.output.compress | both true and false |
| mapreduce.output.compress | false |

Table 3 shows, for each job, the data size at a given stage of the MapReduce IO pipeline as well as the CPU utilization for the HDD-6 setup. The data in the table is normalized with Terasort as baseline. It quantifies the MapReduce job descriptions in Table 2.

### 3.3 MapReduce configurations

Our experiments are run on MapReduce v2 on YARN, which does not have the notion of map and reduce slots anymore. Map and reduce tasks are run within "containers" allocated by YARN. Most of the MapReduce configurations used in our tests come from the defaults in CDH5b1. Table 4 lists performance-related configurations and the values we used. These little-known parameters are often neglected in various studies, including SSD-related prior work [12, 14] and dedicated MapReduce configuration auto-tune systems [10].

Note that these configuration are intended to be performance safe. For each particular customer use case and hardware combination, we expect there is room for further tuning using the values here as a starting point. Further details about MapReduce performance tuning can be found in references such as [19, 18, 17].

## 4 Results

We present the results of our benchmarking in the context of these two questions: (1) for a new cluster, should one prefer SSDs or HDDs of same aggregate bandwidth, and (2) for an existing cluster of HDDs, should one add SSDs or HDDs.

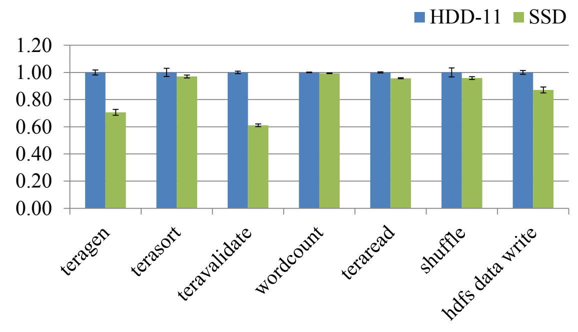
### 4.1 SSDs vs HDDs for a new cluster

Our goal here is to compare SSDs vs HDDs of the same aggregate bandwidth. Let us look at a straight-forward comparison between the SSD (1 SSD) and HDD-11 (11 HDDs) configurations. Figure 2 plots the job durations for the two storage options; the SSD values are normalized against the HDD-11 values for each job. The first graph shows results with intermediate data compressed, and the second one without.

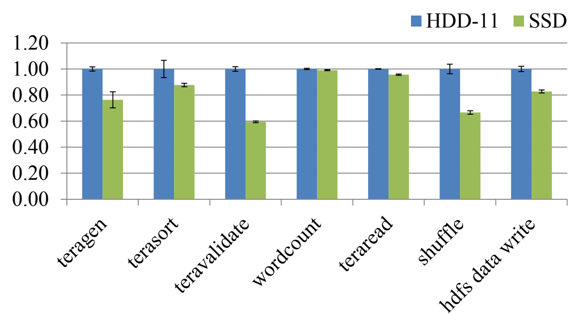
**General Trend.** SSD is better than HDD-11 for all jobs, both with and without intermediate data compression. However, the benefits of using SSD vary across jobs. The SSD related improvement in performance, which is the inverse of job duration graphed in Figure 2, range from 0% for Wordcount to $\tilde{7}0\%$ for Teravalidate and Shuffle.

**Shuffle size determines the improvement due to SSDs.** SSD does benefit shuffle, as seen in Terasort and Shuffle for uncompressed intermediate data. Figure 3 plots the IO sizes of HDFS and Shuffle accesses on HDDs and SSDs. These IO sizes depend on the workload, the operating system, the filesystem and the device being used. Shuffle read and write IO sizes are small compared to that of HDFS and in agreement with our discussion of MapReduce IO patterns earlier.

**Map output compression masks any improvement due to SSDs.** This is evident in the data for Terasort and Shuffle jobs in Figure 2. We believe this is due to shuffle data being served from buffer cache RAM instead of disk. The data in Terasort and Shuffle are both highly compressible, allowing compressed intermediate data to fit in the buffer cache. We verified that, for larger data

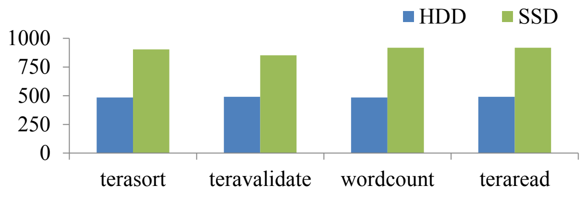


(a) Intermediate data compressed

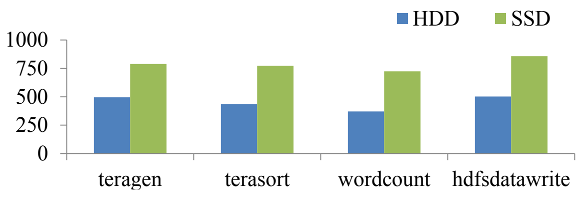


(b) Intermediate data not compressed

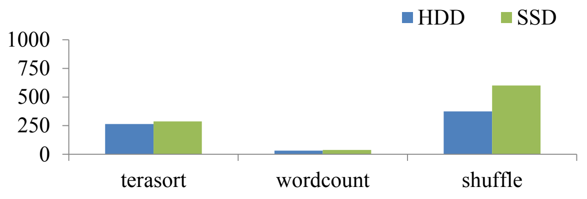
Figure 2: SSD vs HDD. Normalized job durations, lower is better.



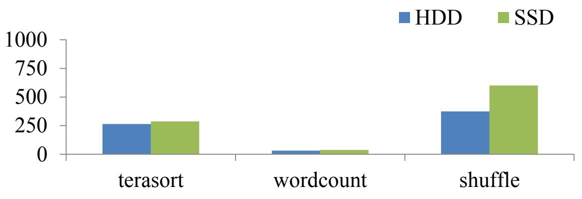
(a) HDFS read IO sizes



(b) HDFS write IO sizes



(c) Shuffle read IO sizes



(d) Shuffle read IO sizes

Figure 3: SSD vs HDD read and write IO sizes for HDFS and shuffle data.

sets (10x), the SSDs give visible benefits (20% faster job duration) even for compressed intermediate data.

**SSD also benefits HDFS read and write.** A surprising result was that SSD also benefits HDFS read and write, as indicated by Teragen, Teravalidate, Teraread, and HDFS Data Write. Turns out that our SSD is capable of roughly 2x the sequential IO size of the hard disks (see Figure 3). In other words, while the application level IO size is "large", the hardware level IO size is determined by the different hardware characteristics. Also, note that these jobs do not involve large amounts of shuffle data, so compressing intermediate data has no visible effect.

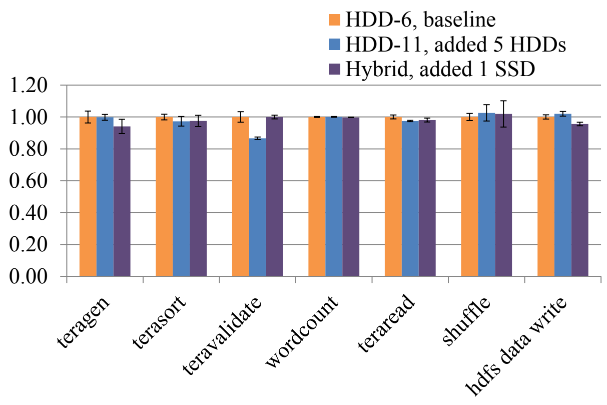
**CPU heavy jobs not affected by choice of storage media.** Wordcount is a CPU-heavy job that involves text parsing and arithmetic aggregation in the map-side combiner. The CPU utilization was higher than that for other jobs, and at 90% regardless of storage and compression configurations. As the IO path is not the bottleneck, the choice of storage media has little impact.

## 4.2 Adding SSDs to an existing cluster

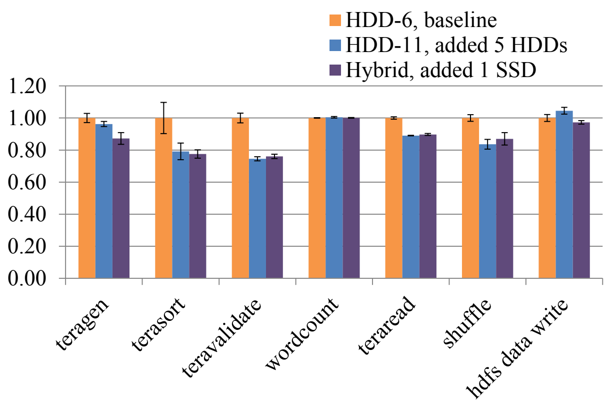
Our goal here is to compare adding an SSD or many HDDs to an existing cluster, and to compare the various configurations possible in a hybrid SSD-HDD cluster.

We use a baseline cluster with 6 HDDs per node (HDD-6). Adding an SSD or 5 HDDs to this baseline results in the Hybrid and HDD-11 setups. On an equal bandwidth basis, adding one SSD should ideally be compared to adding 11 HDDs. Our machines do not have 17 disks; however, we believe the setups we have are enough to give us helpful insights as discussed below.

**Default configurations - Hybrid cluster sees lower than expected benefit.** Figure 4 compares job durations for the HDD-6, HDD-11, and Hybrid setups; intermediate data is compressed in the first graph and not compressed in the second. HDD-11 and Hybrid both give visible improvement over HDD-6. However, even with its additional hardware bandwidth (add 1 SSD vs. add 5 HDDs), the Hybrid setup leads to no improvement over HDD-11. This observation triggered further investiga-

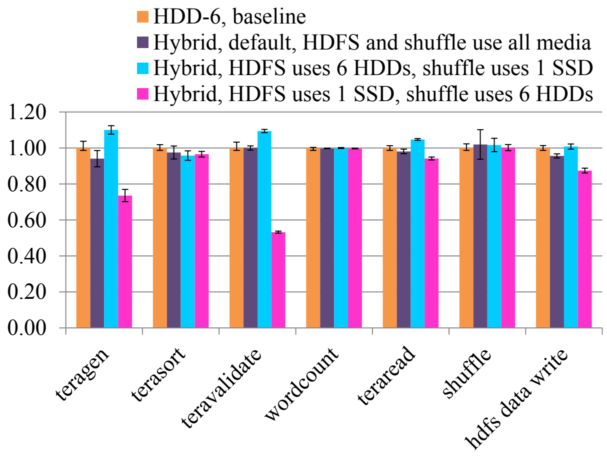


(a) Intermediate data compressed

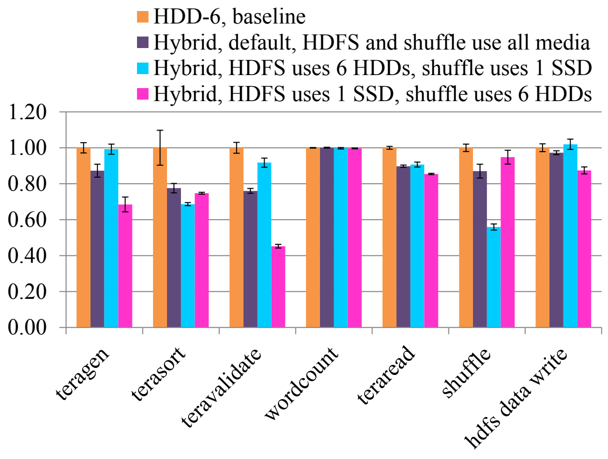


(b) Intermediate data not compressed

Figure 4: Add SSD/HDD to an existing HDD-6 cluster. Normalized job durations, lower is better.



(a) Intermediate data compressed



(b) Intermediate data not compressed

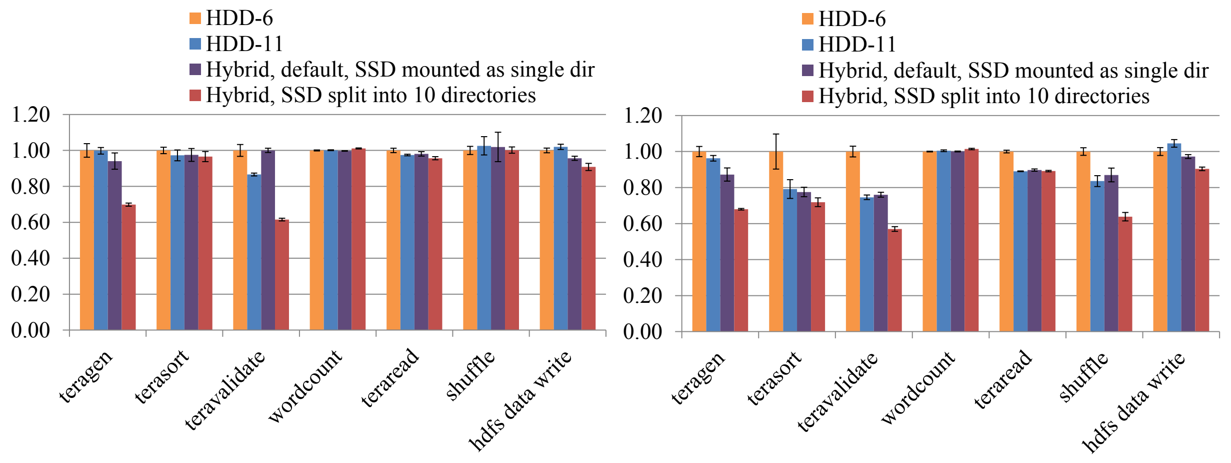
Figure 5: Hybrid modes. Normalized job durations, lower is better.

tions as discussed below.

**Hybrid - when HDFS and shuffle use separate storage media, benefits depend on workload.** The default Hybrid configuration assigns HDDs and SSD to both the HDFS and shuffle local directories. We test whether separating the storage media gives any improvement. Doing so requires two more cluster configurations - HDDs for HDFS with SSD for intermediate data, and vice versa. Figure 5 captures the results of this experiment.

From the results, we see that the shuffle-heavy jobs (Terasort and Shuffle) benefit from assigning SSD completely to intermediate data, while the HDFS-heavy jobs see a penalty (Teragen, Teravalidate, Teraread, HDFS Data Write). We see the opposite when the SSD is assigned to only HDFS. This is expected, as the SSD has a higher bandwidth than 6 HDDs combined. However, one would expect the simple hybrid to perform half way between assigning SSD to intermediate data and HDFS. This led to the next set of tests.

**Hybrid - SSD should be split into multiple local directories.** A closer look at HDFS and MapReduce implementations reveals a critical point — both the DataNode and the NodeManager pick local directories in a round-robin fashion. A typical setup would mount each piece of storage hardware as a separate directory, e.g., /mnt/disk-1, /mnt/disk-2, /mnt/ssd-1. HDFS and MapReduce both have the concept of local directories; HDFS local directories store the actual blocks and MapReduce local directories store the intermediate shuffle data. One can configure HDFS and MapReduce to use multiple local directories, e.g, /mnt/disk-1 through /mnt/disk-11 plus /mnt/ssd-1 for our Hybrid setup. When writing the intermediate shuffle data, the NodeManager picks the 11 HDD local directories and the single SSD directory in a



(a) Intermediate data compressed (b) Intermediate data not compressed

Figure 6: Hybrid with 10 data directories on SSD. Normalized job durations, lower is better.

Table 5: Cost Comparison

| Setup | Cost (US$) | Capacity | Bandwidth | US$ per TB | Cost per performance |
|---|---|---|---|---|---|
| **Disk** | 400 | 2 TB | 120 MBps | 200 | 1x baseline |
| **SSD** | 14,000 | 1.3 TB | 1300 MBps | 10,769 | 2.5x baseline |

round-robin fashion. Hence, when the job is optimized for a single wave of map tasks, each local directory receives the same amount of data, and faster progress on the SSD gets held up by slower progress on the HDDs.

So, to fully utilize the SSD, we need to split the SSD into multiple directories to maintain equal bandwidth per local directory. In our case, SSDs should be split into 10 directories. In our single-wave map output example, the SSDs would then receive 10x the data directed at each HDD, written at 10x the speed, and complete in the same amount of time. Note that splitting the SSD into multiple local directories improves performance, but the SSD will fill up faster than the HDDs.

Figure 6 shows the performance of the split-SSD setup, compared against the HDD-6, HDD-11, and Hybrid-default setups. Splitting SSD into 10 local directories invariably leads to major improvements over the default Hybrid setup.

## 5 Implications and Conclusion

**Choice of storage media should also consider cost-per-performance.** Our findings suggest SSD has higher performance compared to HDD-11. However, from an economic point of view, the choice of storage media depends on the cost-per-performance for each.

This differs from the cost-per-capacity metric ($-per-TB) that appears more frequently in HDD vs SSD comparisons [16, 15, 11]. Cost-per-capacity makes sense for capacity-constrained use cases. As the primary benefit of SSD is high performance rather than high capacity, we believe storage vendors and customers should also track $-per-performance for different storage media.

From our tests, SSDs have up to 70% higher performance, for 2.5x higher $-per-performance (Table 5, average performance divided by cost for the SSD and HDD-11 setups). This is far lower than the 50x difference in $-per-TB. Customers can consider paying a premium cost to obtain up to 70% higher performance.

One caveat is that our tests focus on equal aggregate bandwidth for SSDs and HDDs. An alternate approach is to compare setups with equal cost. That translates to 1 SSD against 35 HDDs. We do not have the necessary hardware to test this setup. However, we suspect the performance bottleneck likely shifts from IO to CPU for our hardware. The recommended configuration is 2 containers per core for MR2, and roughly one container per local directory. On our hardware of 8 cores, having 35 HDDs means either there would not be not enough containers to keep all disks occupied, or there would be too many containers that the CPUs are over-subscribed.

**Choice of storage media should also consider the targeted workload.** Our tests here show that SSD benefits vary depending on the MapReduce job involved. Hence, the choice of storage media needs to consider the aggregate performance impact across the entire production workload. The precise improvement depends on how compressible the data is across all datasets, and the ratio of IO versus CPU load across all jobs.

**Future work.** MapReduce is a crucial component of Enterprise data hubs (EDHs) that enable data to be ingested, processed, and analyzed in many ways. To fully understand the implications of SSDs for EDHs, we need to study the tradeoffs for other components such as HBase, SQL-on-HDFS engines such as Impala [7], and enterprise search platforms such as Apache Solr [3]. These components are much more sensitive to latency and random access. They aggressively cache data in memory, and cache misses heavily affect performance. SSDs could potentially act as a cost-effective cache between memory and disk in the storage hierarchy. We need measurements on real clusters to verify.

Our evaluation here relies on standalone execution of each job on a specific kind of HDD and SDD, that allowed tuning each individual job for optimal performance. We would like to follow this up with an evaluation of multi-job workloads [5, 6] that might present another set of challenges.

Different devices (particularly other SSDs) offer different cost-performance tradeoffs, and it would be interesting to find out the economics there. In particular, SATA SSDs possibly offer different cost-performance characteristics compared with the PCIe SSDs studied here. Researchers in academia are in a good position to partner with different SSD vendors and conduct a broad survey.

Overall, SSD economics involves the interplay between ever-improving software and hardware, as well as ever-evolving customer workloads. The precise trade-off between SSDs, HDDs, and memory deserves regular re-examination over time.

## References

[1] Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org, .

[2] Apache Software Foundation. Apache HBase. http://hbase.apache.org, .

[3] Apache Software Foundation. Apache Solr. http://lucene.apache.org/solr/, .

[4] D. Borthakur. Hadoop and solid state drives. Blog, http://hadoopblog.blogspot.com/2012/05/hadoop-and-solid-state-drives.html.

[5] Y. Chen, S. Alspaugh, A. Ganapathi, R. Griffith, and R. Katz. Statistical workload injector for mapreduce. https://github.com/SWIMProjectUCB/SWIM/wiki, .

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS 2011*, .

[7] Cloudera Inc. Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*.

[9] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *FAST 2014*.

[10] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In *VLDB 2011*.

[11] J. Janukowicz. Worldwide solid state drive 20132017 forecast update. IDC Research Report, 2013.

[12] S.-H. Kang, D.-H. Koo, W.-H. Kang, and S.-W. Lee. A case for flash memory ssd in hadoop applications. *International Journal of Control and Automation*, 6(1), 2013.

[13] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *ISCA 2008*.

[14] J. Lee, S. Moon, Y. suk Kee, and B. Brennan. Introducing SSDs to the Hadoop MapReduce Framework. In *Non-Volatile Memories Workshop 2014*.

[15] J. Monroe and J. Unsworth. Market Trends: Evolving HDD and SSD Storage Landscapes. Gartner Analyst Report, 2013.

[16] PriceG2 Research Report. When Will SSD Have Same Price as HDD. http://www.priceg2.com/.

[17] S. Ryza. Getting MapReduce 2 Up to Speed. Cloudera blog, 2014. http://blog.cloudera.com/blog/2014/02/getting-mapreduce-2-up-to-speed/, .

[18] S. Ryza. Migrating to MapReduce 2 on YARN (For Operators). Cloudera blog, 2013. http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-operators/, .

[19] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

# ParaSwift: File I/O trace modeling for the future

Rukma Talwadker
*NetApp Inc.*

Kaladhar Voruganti
*NetApp Inc.*

## Abstract

Historically, traces have been used by system designers for designing and testing their systems. However, traces are becoming very large and difficult to store and manage. Thus, the area of creating models based on traces is gaining traction. Prior art in trace modeling has primarily dealt with modeling block traces, and file/NAS traces collected from virtualized clients which are essentially block I/O's to the storage server. No prior art exists in modeling file traces. Modeling file traces is difficult because of the presence of meta-data operations and the statefulness NFS operation semantics.

In this paper we present an algorithm and a unified framework that models and replays NFS as well SAN workloads. Typically, trace modeling is a resource intensive process where multiple passes are made over the entire trace. In this paper, in addition to being able to model the intricacies of the NFS protocol, we provide an algorithm that is efficient with respect to its resource consumption needs by using a Bloom Filter based sampling technique. We have verified our trace modeling algorithm on real customer traces and show that our modeling error is quite low.

## 1 Introduction

Historically, benchmarks and traces have been used by system designers for designing and testing their systems. Designing new benchmarks corresponding to new emerging workloads, and getting them approved via standards bodies is a tedious and time consuming process. Thus, system designers usually use workload traces (each scoped to a particular system and application) to validate and verify their designs. However, in order to get a decent representation of an application, one typically needs to capture a trace for a few hours, and for many new emerging applications, this usually leads to the storing of multiple terabytes of data. There have been numerous proposals [6, 13–15] where traces have been leveraged to create workload models that can, in turn, be used to generate I/O patterns. The benefits of modeling traces are: 1) trace models are easier to copy and handle than large trace files 2) using the trace model one can scale the I/O patterns present in the original trace in both temporal and spatial domains and 3) it is much easier to simultaneously and selectively replay respective portions of the trace across respective server mount points using a trace model. It is important to note that a trace model provides a probabilistic representation of the original trace and is not an exact representation of the original trace.

Prior art in trace modeling has primarily focused on modeling block workloads (SAN workloads) [5, 7, 8, 11, 13, 15] and only one attempt had been made at modeling file (NAS) workloads [14] in the context of virtualized environments. In the virtualized environment file I/O workloads from the clients are served by the v-DISK layer of the VM and the underlying NAS storage device only receives the block based read/write requests. The resulting workload does not present many challenges inherent to the file I/O traces like the metadata and the I/O operation mixes, hierarchical file namespaces etc. Hence, modeling these virtualized file workloads is same as modeling SAN workloads. In this paper, we present algorithms and a framework for modeling and replaying NFS workloads that addresses many of the open problems that have been listed in a) the previous NAS trace modeling [14] and replay research efforts [18] b) the limitations of current file system benchmarking tools and c) also some new challenges in trace modeling due to the increased intensity in new types of workloads.

### 1.1 ParaSwift Innovations

ParaSwift makes the following contributions:
**Representing Metadata operations and File System Hierarchies:** Unlike in SAN trace modeling, in NAS trace modeling one has to accurately represent file metadata operations. For example, around 72% of operations in SPECsfs2008 benchmark are metadata related operations. Preserving the order of I/O and metadata operations matters to the application. Operations like creation and deletion of symbolic links, renaming of files/directories can change the underlying file system (FS) image. Accessing file handles via hierarchical

lookups on the file system image has not been handled in previous file system benchmarks. As a result benchmarks either work on a flat namespace or maintain a file to file handle path mapping in the memory. NFS file system benchmarking results are also quite sensitive to the structure of the FS image, the depth and breadth of FS tree, and the sizes of the various files and directories.

**Workload Scaling:** Workload scaling requires understanding and representing logical file access patterns so that workload scale-up operations can increase the load on specific files which might not be physically co-located on the storage media. Thus, there is a need to be able to observe the load intensity characteristics of the workload, and model it accordingly. Similarly, one needs to take the spatial and temporal scaling parameters as input, and subsequently replay using the trace model to satisfy the scaling requirements. Most of the file system benchmarks do not allow for this level of control with respect to scaling operations.

**Handling New Workloads :** Increasingly, new types of transaction intensive workloads are emerging where millions of transactions get executed per second (e.g. mobile ad applications, stock trading, real-time supply chain management etc). Storing file system traces at this scale would result in extremely large trace files. There are challenges with the amount of memory and compute resources required to 1) capture the stateful nature of the client-side protocol and 2) to learn the intricacies of a large number of file handles encountered in the trace and their off block boundary access size granularities. Trace model building algorithms that have been articulated in prior art [5,7,11,13] were not designed for handling these challenges. For instance, there is a minimum of 50x increase in workload intensity and number of clients per trace from the Animation workload traces captured in 2007 [12] vs. the one captured by us at our customer site in 2013. Hence, it is desirable to be able to dynamically create models as traces are being captured to reduce the resource requirements of tracing frameworks.

**Lack of Similarity in Underlying File Systems:** Usually, users of traces assume that traces collected in one setup (file system) can be used on a different file system. For example, the file mount parameters, read/write transfer sizes, and age of the file system can have substantial impact on the load being seen for the file server. Thus, it is important to leverage file traces in conjunction with the characteristics of the underlying file system. Apart from overheads of storing traces for on-demand replays, the existing trace replay tool [18] suffers from two other shortcomings 1) Replay timing accuracies suffer as the number of clients and NFS mount points in a single intense trace runs to hundreds (E.g. 640 client/server IP combinations in Animation trace of Table 1 on Page 9) 2) even under the same client workload, it is possible that different file servers based on the same protocol produce very different traces. Hence, replay of traces captured on one setup and replayed on another without inspecting the responses would not be useful.

In this paper we present a framework and algorithm called *ParaSwift* that can model file system traces (NFS), and it addresses the concerns that have been listed above. At this moment ParaSwift can be used to model and replay both NFS v3 and v2 protocols. The modeling code can be also used to model SAN traces but in this paper we are specifically focusing on NFS trace modeling.

## 2 System Overview and Algorithms

Figure 1 (Page 4) illustrates ParaSwift's model building architecture. One of the key design principles behind our thinking is that we need to be able to build models for very large traces in a resource efficient manner. This, in turn, forced us to re-think the steps involved in trace model building pipeline. ParaSwift's model building process is divided into 5 distinct phases:

**Phase 1: Trace parsing :** trace parser extracts and presents each NFS request and its corresponding response in the order of the request arrival to the trace sampling component.

**Phase 2: Inline trace sampling :** as a NFS request-response pair is streamed through ParaSwift, corresponding model building data structures have to be updated in-memory. Over a period of time the host memory will get exhausted and the respective data structures will spill over to the disk. This, in turn, will throttle the inline stream of NFS request-response processing. The inline trace sampling component helps in reducing the probability of the data structure spill over by discarding the redundant I/O's.

**Phase 3: Inline trace model building :** as the request-response pair is streamed through ParaSwift, in-memory data structures representing the various aspects of the trace are updated. The data structure allocation process has been designed to 1) make most efficient use of the host memory 2) eliminate the need for doing multiple passes through the trace for model building 3) represent the workload characteristics in a very succinct way without much information loss.

**Phase 4: Batch trace processing :** the in-memory data structures constructed by ParaSwift cannot be directly fed to a load regenerator. Few optimizations need to be done on the data structures for extracting the necessary model parameters. These procedures are essentially batch in nature.

**Phase 5: Trace model translation for replay :** For the purpose of workload replay we use a commercial load generator called *Load DynamiX*$^{TM}$ [2]. *Load DynamiX* allows us to achieve a high fidelity translation of the

ParaSwift trace model to an equivalent workload profile. *Load DynamiX* also provides us with the ability to seamlessly scale the trace workloads to over a million clients per NFS server mount point.

In order to understand Phase 2 of the pipeline, one needs to understand Phases 3 and 4. Hence, below, we discuss Phase 2 after we discuss the other phases of the model building pipeline. We discuss each of the phases in the modeling pipeline in the following sections respectively.

## 2.1 NFS Trace Input and Parsing

**NFS Trace :** NFS protocol consists of two categories of operations. I/O operations that directly access or manipulate the contents of a file (NFS reads and writes) and metadata operations which read or write the file/directory metadata like getting file/directory access permissions (getattr), making file links (link, create link), setting directory/file ownership info (setattr) and updating filesystem information (fsinfo/fsstat). NFS traces are usually captured using OS tools like TCPDUMP in Linux and stored with a .PCAP or a .TRC extension. Since NFS traces tend to be bulky, efforts have been made to store them more efficiently in a compressed format known as the DataSeries, introduced in [4] and recommended by the Storage Networking Industry Association (SNIA). NFS trace stored in a DataSeries format (.ds) is essentially a database with all the information about the various operations (I/O as well as metadata) along with the corresponding network IP/port as well as the respective RPC packet timing information.
**Trace Capture Tool :** We have developed a tool within our group to capture NFS traces over network by mirroring network switch ports. This tool can handle rates of up to 16 GBps. For storing the NFS packets, the trace capture tool leverages the DataSeries software. Each incoming NFS packet representing a request or a response to a particular NFS operation is parameterized, compressed and stored as a separate entry in the corresponding operation specific DataSeries table for which the schema is predefined by the tool. The library stores the timestamp of each request/response in the order of their arrivals along with the corresponding network information in a separate DataSeries table. Trace replay is not part of this tool.
**Trace Parser :** ParaSwift cannot directly work on the compressed .ds traces. In order to extract a model, it has to un-compresses the respective DataSeries tables, pair the corresponding request/response operations, and reconstruct the temporal order of the requests. As we shall see later, pairing requests with the corresponding response helps in trace model correction. We have written approximately 350 LOC in DataSeries software distribution to accomplish the parsing and to generate an equivalent comma separated file (.csv).

Each line in the .csv file represents a request and its corresponding response along with the operation specific attribute values. The process of parsing a .ds file is called as Merge Join in ParaSwift as it's a join between the respective operation table and the table storing operation's timestamp and the network details in the DataSeries. Merge Join process is very light weight on memory as it operates on one row of each DataSeries table at a time and generates one request response pair at a time.

## 2.2 Inline Model Extraction

Each unsampled request/response pair extracted by the ParaSwift parser flows through all the components of the ParaSwift's inline processing pipeline as shown in Figure 1. We refer to these steps as inline, as they update in-memory data structures as the I/O's are streamed through the system.

### 2.2.1 Workload Stream and Pattern Separation

Due to host and storage virtualization, a NFS trace could contain requests coming in from multiple network clients going to multiple NFS mount points (identified by a host IP). Each distinct client-host IP combination is referred by ParaSwift as a stream, which is modeled separately and replayed simultaneously. This capability is not present in any of the benchmarks/replayers to date. Secondly, segregation of individual operation requests into macro level patterns per stream is essential from the user application perspective and hence the order of operations is more important than the proportions in which the various operations land on the NFS server.

Hence, this component of ParaSwift focusses on two aspects: 1) separate each request into a stream identifier based on its client and host network IP and 2) identify and separate individual operation requests per stream into macro level workflows called the workload patterns.

A workload pattern *j* belonging to a stream *i* ($WP_{i,j}$) is extracted based on: 1) maximum idle time between two consecutive request arrivals 2) maximum time elapsed between the previous response and the new request and 3) file handles involved in the two consecutive requests within a stream. For the two operations to be a part of the same workload pattern we do not mandate that they have to operate on the same file handle. For example, a physical client executing a Linux *make* task would involve multiple files. Each workload pattern extracted by ParaSwift is replayed separately during regeneration. The length and life-time of a workload pattern is

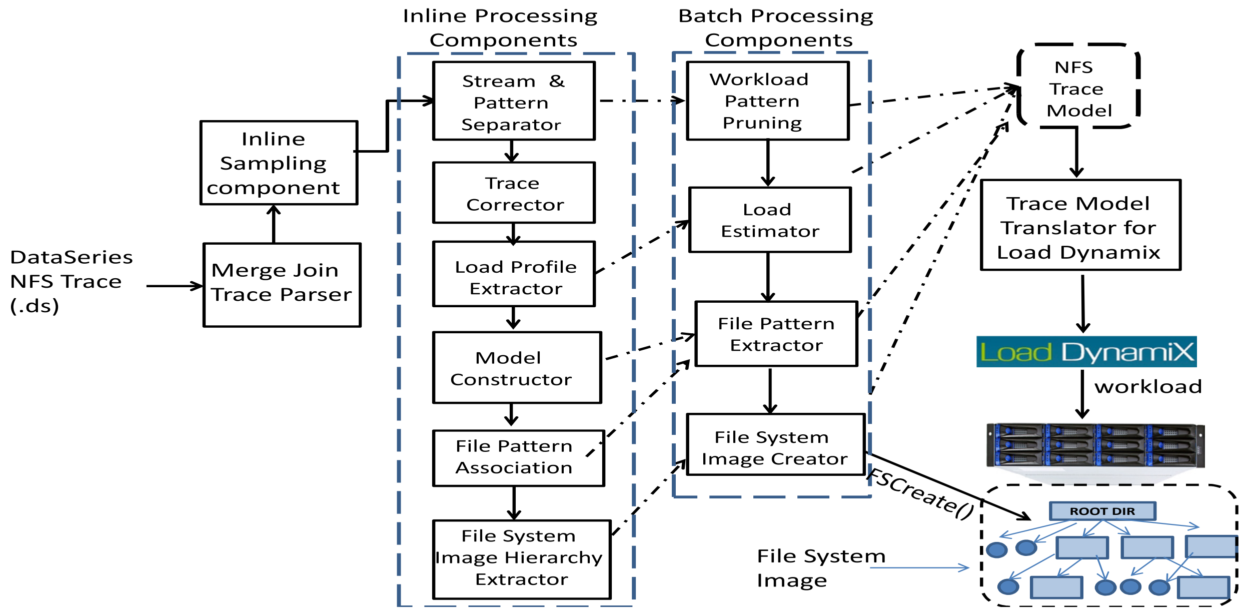

Inline Processing Components
Batch Processing Components
DataSeries NFS Trace (.ds)
Merge Join Trace Parser
Inline Sampling component
Stream & Pattern Separator
Trace Corrector
Load Profile Extractor
Model Constructor
File Pattern Association
File System Image Hierarchy Extractor
Workload Pattern Pruning
Load Estimator
File Pattern Extractor
File System Image Creator
NFS Trace Model
Trace Model Translator for Load Dynamix
Load DynamiX
workload
FSCreate()
ROOT DIR
File System Image

Figure 1: ParaSwift Architecture

recorded appropriately and exercised via specific knobs provided by the *Load DynamiX* replayer.

### 2.2.2 Trace Correction

One another aspect of trace modeling and replay for NFS workloads, is dealing with errors or I/O information losses. Whether the operation was missing or was mistakenly issued by the client can be inferred to a large extent from the corresponding NFS response. This is why a request and its response are presented together to the inline processing phase by the ParaSwift parser. In ParaSwift, we made a conscious decision to either correct the NFS request errors or to discard the erroneous operations without contaminating the final workload model.

If ParaSwift observes two successful consecutive remove requests on the same file handle, then in between the two remove requests there needs to be a create, rename, symbolic link, or link (hard link) request. If two operations cannot appear consecutively, we correct by additional operation insertion or remove the later request (if it was a failure). For operation insertion we too follow a table-driven heuristics approach similar to [18]. ParaSwift's trace correction scope is restricted within a workload pattern. Trace correction happens per request and with respect to the previously parsed request. A workload pattern is registered only post correction.

### 2.2.3 Load Profile Extraction

Each of the previously derived workload patterns need to be associated with load profiles. It means that streams which are more intense and perform operations at a faster rate than others need to be represented appropriately. Secondly, some workload patterns might be more bursty. The reasons for this behavior could be that some patterns might be script driven (e.g. Linux make task), whereas, some others might be driven by humans (e.g. reading a balance sheet). Based on these requirements, ParaSwift updates the following two parameters inline: 1) request arrival rates per stream ($C$) and 2) request arrival rates per workload pattern per stream ($r_{i,j}$).

Average request inter arrival time ($RIT_{i,j}$) in a workload pattern $j$ of stream $i$ is derived inline every 5 seconds interval. Based on this the total number of NFS operations that would be generated ($r_{i,j}$) per interval, by a given workload pattern, is obtained. ParaSwift also computes 80$^{th}$ percentile value of the max number of NFS operations seen in the storage system from a given stream per interval, referred to as $C$. These computations are revisited and refined during the batch phase.

### 2.2.4 Model Constructor

In ParaSwift architecture we separate the process of workload pattern identification from its storage. Workload patterns have to be registered in a memory efficient data structure. We could have saved each pattern separately as identifier separated operation entries. However, the workload patterns could be many in number and there could be many details associated with the pattern like: the file handles used and the attribute values of the operations they contain. Thus, a lot of memory can be needed to represent these details.

We chose to represent each association between a pair of operations via a shared adjacency matrix structure per workload stream. There are about 21 different operations (including I/O and metadata) that ParaSwift recognizes with a unique index based on the NFS v2/v3 standards. If an operation with index $y$ follows an operation with index $x$ in the workload pattern $WP_{i,j}$, the corresponding entry in the adjacency matrix is incremented by one. In order to allow us to reconstruct a workload pattern from a matrix we also build two additional data structures: a workload pattern start list and an end list per stream. These lists contain operations with which the workload patterns can begin or end respectively with the corresponding probabilities. All these data structures are updated inline and the normalization of probabilities is done in the batch phase. Figure 2 describes some of these data structures. Figure 2 also points to additional datastructures like the *fileHandleMap* used to account for the unique file handles accessed, and how they are linked with each other.

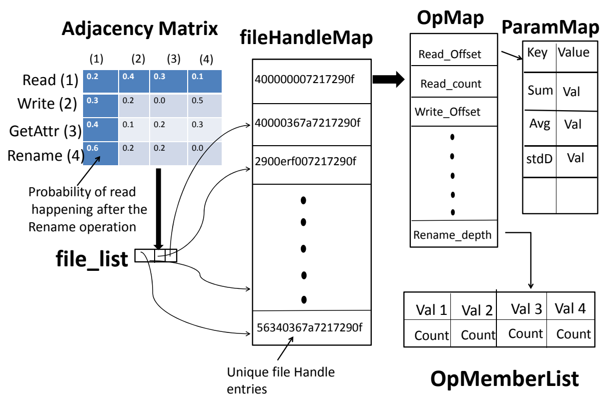


Figure 2: In Memory Data Structures in ParaSwift

### 2.2.5 File pattern association

The values of attributes for a particular NFS operation varies significantly even for a given file handle. For instance a file may not be always read from the same offset and for exactly the same number of bytes. Hence, it makes more sense to model them as distributions. The total number of unique files accessed and the type of operations performed on them is a huge set. Capturing these details in-memory was yet another challenge for ParaSwift.

Values of various attributes per NFS operation and the per file handle is recorded by ParaSwift in a memory resident data structure called the *OPMemberList*, which records each unique value of the attribute and the number of times such a value was reported in the trace separately. These values are later fitted to a distribution function per file handle and per operation, and written to the disk during the batch phase. In order to speed up processing, and reduce the number of compute passes over the data structures during the batch phase. A superset of parameters needed to compute the various distribution parameters like distribution averages are also partially computed inline and stored in a data structure named *ParamMap* as illustrated in Figure 2.

### 2.2.6 File System Image Exaction

ParaSwift reconstructs the essential File System (FS) hierarchy for the workload replay as-is from the trace. Generating FS image also happens inline and is stored in-memory. Since complete information about the exact location of the directory or a file in the FS may not be available till the end of parsing, we differ writing the actual FS image to the NFS server until the end.

We use a small in-memory representation called the *fstree* similar to the one used in [16] which stores just the right amount of information to create an FS image later. fstree contains pointers to file and directory objects. Each directory object only contains a pointer to its parent directory and its child directory/file objects and no other information. Each file object only contains information on its latest size in bytes obtained from the NFS operation response packet. During FS creation, the file size parameter is used by ParaSwift to create a file with equivalent number of bytes filled with random data pattern. We do not store directory/file i-node metadata such as access permission, ownership info etc. So far we do not use this information during replay.

## 2.3 Batch Model Processing

There are some aspects of trace model extraction which need to have a macro picture of the overall statistics in the trace. Such decisions cannot be made inline and have often necessitated multiple passes through the trace in the past [6, 11, 13]. We instead do multiple random accesses through few of the partially processed data structures built in-memory like the *OpMemberList* and *ParamMap* in Figure 2. These components are listed in Figure 1 and we discuss them in this section.

### 2.3.1 Workload Pattern Pruning

During the inline model extraction phase a large number of workload patterns are mined. This could be partly because these patterns are obtained heuristically. However, depending on the frequency of request arrivals in the workload pattern, not all of them might be significant enough to be regenerated.

ParaSwift defines a $Path\_Probability\_Threshold$ which is the minimum expected probability of occurrence of a workload pattern for it to be included in the final model. The Path probability ($s_{i,j}$) of a given workload pattern is the cross product of start probability of the first operation (in the start list) in the workload pattern with the subsequent transition probabilities of other operations obtained from the adjacency matrix until an operation which also appears in the end list is encountered.

**Algorithm 1** Workload Pattern Pruning Algorithm

```
for all Operation index entry in the start_list do
    Add the entry to IncompletePathsList sorted by the descending order of the start probability
end for
while $exitcondition == false$ do
    BestPath = Get highest probability path from IncompletePathsList
    x = index of the last operation in the BestPath
    for all y in $AD_{i,j}[x][y]$ do
        if $AD_{i,j}[x][y].count$ != 0 then
            BestPath.nextOp(y)
            BestPath.pathProbability $*=$ $AD_{i,j}[x][y].prob$
            if $end\_list\_i.contains(y)$ & BestPath.pathProbability $>$ Path_Probability_Threshold then
                completePathsList.add(BestPath)
                totalProb = totalProb + BestPath.pathProbability
                M = M + 1
            else if ! $end\_list_i.contains(y)$ & BestPath.pathProbability $>$ Path_Probability_Threshold then
                IncompletePathsList.addSorted(BestPath)
            end if
            if totalProb $>=$ Coverage_Threshold || M $>=$ Total_Paths_Threshold then
                exitcondition = true
            end if
        end if
    end for
end while
```

Since adjacency matrix can also be realized as a directed graph, ParaSwift uses a breadth first search algorithm as listed in Algorithm 1 with greedy path pruning for extracting the most significant workload patterns. Every incomplete pattern is inserted into a sorted path list. The path with the highest probability is picked first and expanded. The algorithm terminates when either the total number of workload patterns extracted exceeds $Total\_Paths\_Threshold$ (presently 25) or sum of the path probabilities for the paths extracted exceeds $Coverage\_Threshold$ (presently 70%). These parameters are tunable by the user based on the expected fidelity of the outcome.

### 2.3.2 Load Estimator

The two quantities, load intensity per workload pattern ($RIT_{i,j}$) and intensity per stream ($C$) calculated in the previous phase are based on averages, and hence, there is a possibility of under-loading the system. Secondly, workload pattern pruning step also eliminates some of the less frequent workload patterns which may further reduce the load intensity of the stream. During replay, in order to achieve a realistic load factor, ParaSwift utilizes two additional load related knobs provided by *Load DynamiX*. These include: concurrency factor per workload pattern ($p_{i,j}$) and total number of simultaneous clients per stream in the system ($N$).

With $s_{i,j}$ representing the path probability of a workload pattern $WP_{i,j}$ and $M$ being the total number of significant workload patterns per stream $i$ obtained from the previous step, values of $p_{i,j}$ and $N$ are then computed as:

$\sum_{i=1}^{M} s_{i,j}$ ¡= 1 by the law of probability -(1)
$p_{i,j} = s_{i,j} \times N$ for i=1,..M - (2)
$N = \frac{C}{\sum_{i=1}^{M}(s_{i,j} \times r_{i,j})}$ - (3)

All of the load profile related parameters are represented at a granularity of 5 seconds over the entire period of observation. Any change in the values of these parameters across intervals is expressed as a corresponding ramp up/ramp down factor by ParaSwift during the Phase 5 of the pipeline illustrated in Figure 1.

### 2.3.3 File Pattern Extraction

In this step ParaSwift performs a function fitting per NFS operation attribute per file handle. To regenerate a value for an attribute (e.g. offset and I/O size for a read/write), instead of regenerating it empirically as in [14, 17] from the $OpMemberList$ data structure ParaSwift does a least error function fitting of the parameters. ParaSwift is aware of a few standard distributions like the Constant, Gaussian, Normal/Uniform, Log normal, Exponential and Polynomial. The least error function is chosen and the corresponding function parameters are extracted.

#### 2.3.4 FS Image Creation and Hierarchical Namespace Lookups

ParaSwift reads the fstree created during the inline step and generates an appropriate FS image on the NFS server. Emulating a real client using benchmarks is a tough problem. NFS access begins with a lookup for the file handle in the correct FS directory path. This requires the client to be aware of its present path in the File System mount point. This could be accomplished by maintaining a file to file handle path map during replay. However, this technique does not scale as the working sets expand. To address this problem, ParaSwift creates every file seen in the trace at an appropriate location along with the corresponding soft link at the ROOT of the mount point during the fscreate() procedure. Load regenerator works on the links, but this ensures that every link read for a write operation translates into corresponding lookups and reads/writes for the actual file in the NFS server. However, few operations like file delete would still leave the actual file untouched during replay.

**Scaling :** ParaSwift provides an option of spatial scaling during the FS image creation. One can specify scaling of following two types: a) uniform scaling b) proportionate scaling. In uniform scaling ParaSwift simply adds more files in every logical path by a factor provided as an input. Associated soft links are also created accordingly. Proportionate scaling is the more interesting option provided by ParaSwift. ParaSwift at present supports few options like, ”scale up metadata portion by x%”. In this case ParaSwift already knows through OpMap (in Figure 2) scan processing which file handles are accessed for metadata versus which are accessed for I/O operations. Accordingly, it scales up the file count in the appropriate directories. For time scaling, respective load profile parameters are proportionately increased/decreased by ParaSwift

## 2.4 Trace Model Translation for Replay

Problem of workload regeneration is often delinked from its modeling. However, delinking the two aspects often results in gaps as discussed in section 1.1. We use *Load DynamiX* as a workload replayer in ParaSwift at the moment. *Load DynamiX* allows us to 1) emulate the probabilitic ordering of the various NFS operations within a workload pattern; 2) regenerate the various attributes of a NFS operation like NFS read size and offset etc. using Load DynamiX offered statistical functions; 3) scale infinitely; 4) exposes excellent knobs with respect to controlling request intensities; 5) does a time accurate request regeneration; 6) and supports i-SCSI (SAN), NFS and CIFS protocols. We use version 30.21444 of *Load DynamiX*.

*Load DynamiX* allows ParaSwift to specify per NFS operation specific attribute values as either averages or enumerations per file handle. Averages are not good enough and enumerations would not scale for larger traces. *Load DynamiX* additionally provides an ability to specify any distribution as a function of *Random* distribution function. Given a max and a min value, the Random function generates a random value within the range.

Based on the theory [10], each realization of a Random function would result in a different function. ParaSwift maps various distribution functions and their parameters to appropriate min/max values of the Random function to regenerate the respective distribution function.

*Load DynamiX* provides a Perl API module for creating a NFS workload project. The APIs support functionalities like 1) workload pattern creation; 2) respective load profile specification; 3) associating individual operations within a workload pattern to appropriate file handles, and operation parameters to an equivalent Random distribution parameters. We have written additional 700 LOC in *Perl* to leverage the *Load DynamiX* Perl module and generate an equivalent workload project by directly reading from ParaSwift workload model. Workload project can be directly imported via the *Load DynamiX* client GUI and replayed on the NFS server on which the FS image has been already created.

## 2.5 Inline Sampling

Our thinking behind building in-memory data structures is to speed up the trace processing times. We already see about 50x-100x increase in the request intensity between animation workloads captured in the year 2007 and those captured by our internal tool in 2013 listed in Table 1.

Workloads on the other hand may not change often. A trace data collected for first 5 minutes might be well representative of the later 10 minutes. If we could infer and exploit such self similarity in the traces *inline*, we can retain in-memory processing benefits at higher trace intensities. ParaSwift implements a novel inline content aware I/O sampling technique.

The technique is based on the concept of Bloom Filters (BF) [9]. BF is a space efficient, probabilistic data structure used to test containment of an object in the set. Efficiency of the implementation depends on the trace parameters used to test the containment. Our implementation is based on the following design constraints:

**1. Intensity of I/O’s needs to be retained :** Sampling may make it appear that incoming I/O stream is very slow. ParaSwift model needs to preserve the original I/O

intensity.
**2. Preservation of operation distributions :** Filtering should not introduce a skew in the NFS operation distributions.
**3. Compute and Memory Footprint :** Data structures for containment have to have a low memory footprint and allow a quick lookup.

### 2.5.1 The Technique and Implementation

For the purpose of establishing the need for sampling, we merged several subtraces obtained from the IT workload trace capture to be discussed in section 3 as in Table 1. It took ParaSwift 4 hours to process a trace (.ds) worth 20 GB. We profiled this execution and learnt that the processing times could have be slashed by reducing the footprint of the model building data structures which were being extended to the disk as the number of unique files in the trace changed too often. We realized that it was not the number of unique files but the unique characteristics that matters for modeling. We sampled out the data at 60% (discarding 60%) with our technique and were able to complete the task in 1 hour with only 10% loss in fidelity.

For read and write operations, it is often important to look at the specific properties rather than just looking at the file handle. For instance, a random small read vs. a large sequential read, a file overwrite vs. partial write would have completely different performance implication. Hence, we look at the offset and the size characteristics of a read/write as the distinguishing factor. For metadata operations ParaSwift does not worry about the exact values of the attributes being set on the file or the directory i-nodes. In some cases like the rename or the create link operations the relative path or the location of the target with respect to the source file handle matters (from the file system performance point of view).

We adopt BF's only for detecting redundant read/write operations. BF lookup based on the containment function gives out a location index. If the corresponding bit index is already set, this implies that the value might already be there. BF can at times give false positives. However, there are no false negatives. Our design constraint is to minimize number of false positives (fp) for read/write operations.

We use BF fundamentals to decide: $m$, which is the number of bit locations a given function would hash to; $k$, which is the total number of hash functions we should use for a set of $n$ input values. This is analogous to number of I/O's a trace would contain (each request-response pair being one I/O). We fix our fp rate to 10% which also lowers the bounds of the final model fidelity. We could reduce fp rate further, but we are convinced that its effectiveness depends on the variance of the parameters to be hashed in the trace, which are assumed to be uniformly distributed by the BF theory, and may not hold true for every workload trace.

ParaSwift uses a partitioned BF [9]. Our first hash function hashes offset and the second function hashes the size of a read. Similar BF's are designed for the write operation as well. Both of the containment functions work irrespective of request file handle. Semantically this implies that if a request makes access to a unique offset within a file and for a unique number of bytes, it is a new type of I/O irrespective of its file handle. This comparison indirectly favors I/O's which access files of non-identical sizes.

When sampling is turned on, the trace parser generates a random number per I/O to be processed. If the random number is within the expressed sampling rate, the request/response pair is presented to the sampling component. Whether an I/O is to be sampled out or not is decided by the BF for a read/write operation. For the metadata operation, only if it has not been recorded previously in adjacency matrix it is retained, else it is discarded. If it is a metadata operation to be discarded like rename, and create operations, ParaSwift records the relative path distance, as discussed earlier, that is applied during the replay. For all operations, the corresponding adjacency matrix is always updated irrespective of the verdict to avoid an operation distribution skew. Also, the number of unique file handles per interval are recorded, though the corresponding file handle maps as seen in Figure 2 are not updated if the I/O is to be sampled out. This preserves the working set sizes. During model translation, appropriate number of pseudo files are created to regenerate the original working set.

The two (read and write) partitioned BF's each with two partitions account for 722 KB of total memory space. Space needed for target and source path distance accounting, for the metadata operations, is negligible. Both the hash computations and filter lookups are constant time operations. This makes sampling strategy pretty amicable to inline trace model building. We use a popular and a reliable Murmur hash function [1] for all the Bloom filters.

ParaSwift, excluding the model to replayable project translation part, has been completely written in $c++$ with approximately 2500 LOC.

## 3 Evaluation

In this section we evaluate ParaSwift against 3 different NFS traces. We compare the accuracy of ParaSwift in recreating various workload patterns/characteristics observed in these workloads. We verify whether ParaSwift can: A) recreate various operations (I/O as well as metadata) seen in the original trace in right proportions; B)

Table 1: NFS traces used for Validation

| Label | Capture duration | Number of unique streams | Total capture size |
|---|---|---|---|
| IT | One week | 114 | 22 TB |
| Cust1 | One weeks | 63 | 17 TB |
| Animation | 13 Hours | 640 | 1.7GB |

recreate read/writes with appropriate file offsets C) have right I/O size distributions; D) emulate right file working set sizes with appropriate file access popularity distribution E) mimic identical read and write payloads to the NFS mount point per stream and finally F) to retain all of these capabilities while sampling the input trace at variable frequencies. As the techniques used in this paper are statistical in nature our goal in this section is to be as close to the original workload statistically and not identical. We shall refer to the above itemized list of metrics while presenting the experiment results (in the same order).

## 3.1 The Workloads

Table 1 lists the NFS traces used to validate ParaSwift. Traces labeled as IT and Cust1 are collected by our internal trace capture tool for a week's duration. IT being our internal IT workloads and Cust1 representative of a customer running version control and scientific simulation like workloads concurrently on separate NFS mount points but as a part of the same trace. Each subtrace (a portion of the larger trace) is less than 4 GB and depending on the I/O intensity, covers a duration anywhere between 8 to 15 minutes. There are 900 such subtraces for IT and 495 for Cust1 trace covering a one week duration. We also regenerated a subtrace (1 out of 473) from the Animation workload traces (*Set 0*) captured in 2007, and published at the SNIA site [12], described in Table 1. Detailed information about their capture and analysis can be found at [3].

Each of these traces had multiple clients and NFS mount points. ParaSwift extracted each combination as a separate stream, modeled the various streams in the same run of the ParaSwift and regenerated them simultaneously via *Load DynamiX*. Each trace was run for the same duration as the original trace. Having the capability of being able to emulate multiple streams simultaneously is an important contribution of ParaSwift. We quantify the accuracy of regenerations on a per stream basis.

## 3.2 System Configuration

ParaSwift runs on a Linux based VM with 15 GB memory extensible up to 64 GB with 100 MBps network connecting a NFS storage server. Load regeneration was accomplished using 1 GBps full duplex physical port of *Load DynamiX* which could scale to over a million virtual ports (streams). Since ParaSwift does not focus on reproducing the identical performance profile (latency, file system fragmentation) we use a standard storage server configuration with 6 GB memory, 1 TB volume with 7500 rpm disks and 1 GBps network for all trace regenerations.

We validate ParaSwift by comparing the original workload trace with regenerated workload trace captured by *Load DynamiX* at the time of its execution.

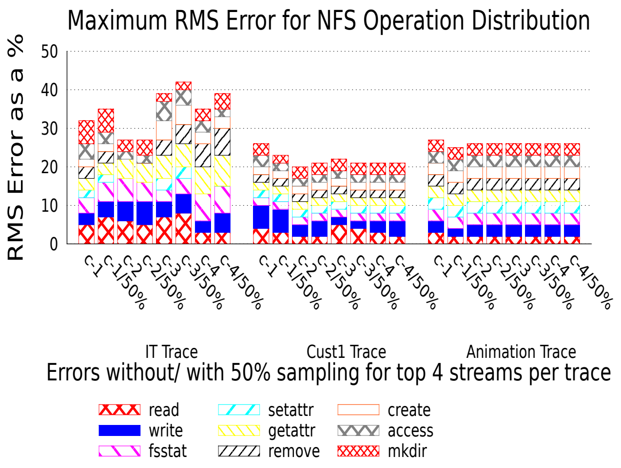
## 3.3 Experiments

In this section we present evaluation results based on three traces: IT, Cust1 and Animation as described in Table 1. We perform per metric statistical comparison of the original trace with the *Load DynamiX* collected synthetic trace for all of the earlier mentioned dimensions. We represent error as the Root Mean Square (RMS) error which is the standard deviation of the differences between regenerated values and original values of a metric in its respective distribution buckets.

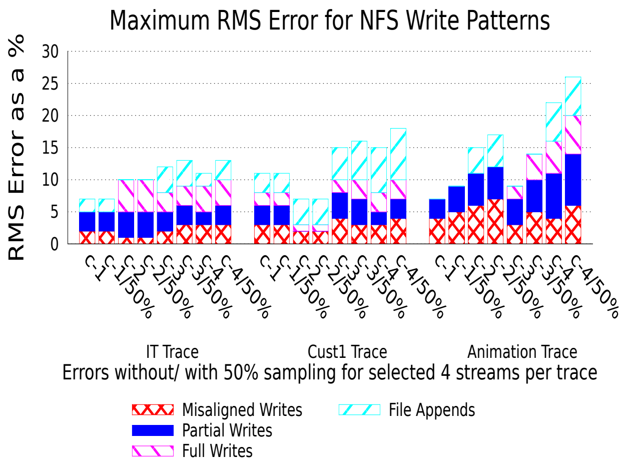
### 3.3.1 Choosing subtraces

Since the total number of subtraces collected for IT and Cust1 workloads is huge, for the purpose of validation we sampled 20 subtraces belonging to different durations of the day and week. For Animation workloads we modeled one subtrace out of the 473 published subtraces. We compute metric errors per stream in a subtrace separately. In this paper we report results for the top 4 streams in a single subtrace for all three workloads. However we have validated that the various metric error bounds hold true for other streams in the three traces as well as for the other randomly chosen 19 subtraces for IT and Cust1 workloads. According to our preliminary analysis of 3 workloads in Table 1, workload patterns for these different traces varied quite a bit. Therefore, even though we only had 3 traces, we actually had many subtraces with vastly different characteristics.

**A)** Figure 3(a) represents the RMS error for distributions of various operations for chosen streams in the each of the three traces. Streams which 1) had the highest request rate for chosen capture interval and 2) demonstrated wide variation in the type of NFS operations issued were chosen. For each stream (denoted as symbol C in the graphs) we also compared the corresponding errors when the original trace (.ds) was sampled at 50% (denoted as C-50%) of the actual trace size using ParaSwift's Bloom Filter (BF) technique. We see that max RMS error for all of the streams for all the metrics in Figure 3(a) is below 10%. Also there is not much dif-

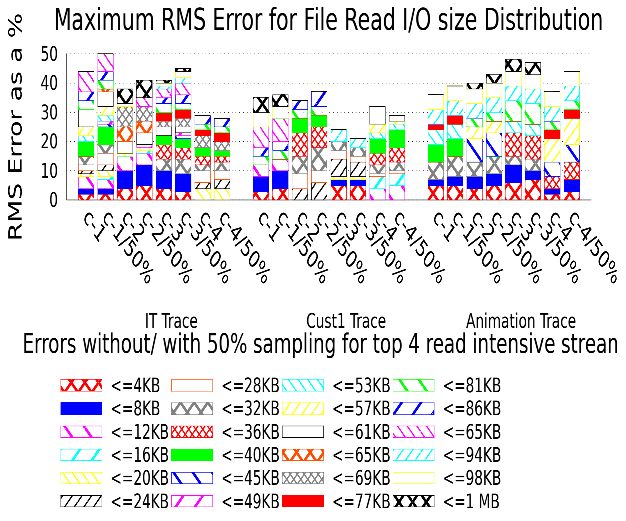


(a) I/O and Metadata Operation Distribution Errors

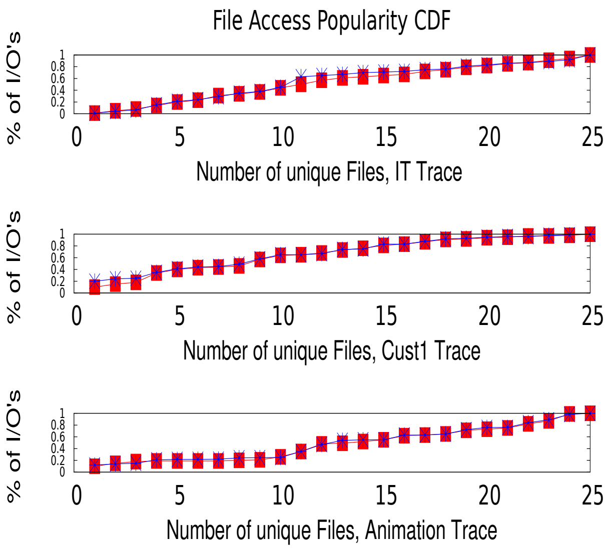


(b) Write Pattern Distribution Errors

Figure 3: RMS Error for various distributions for the top 4 streams of each of the 3 workloads



(a) Read I/O size Distribution Errors



(b) File Access Popularity CDF

Figure 4: RMS Errors for Read I/O size Distributions and CDF of File Access Popularity

ference in the RMS errors between a sampled and an unsampled version of the trace model. Low errors in operation distribution are attributed to the Markov chain representation (adjacency matrix) and later the smart path pruning algorithm constituting ParaSwift's batch model processing phase. Also note that the the overall value of the y axis for each bar graph is a cumulative sum of the individual RMS values.

**B)** Figure 3(b) lists the RMS error for a distribution of various write patterns regenerated by top 4 streams at varying file offsets per a workload trace. Any read/write which does not happen at the 4 KB block boundary is misaligned for us. Any write which starts at offset 0 and ends at offset equal to file size prior to the operation (obtained from the response of the operation) is an overwrite. Any write which begins at the end offset of a file is an append. Any write not satisfying the latter two criteria is a partial write. For this comparison to be effective, streams which were intense as well as had a high fraction of writes were chosen. As seen in Figure 3(b) ParaSwift is able to fairly mimic the various write patterns restricting the RMS errors within 10%.

**C)** We also computed RMS errors for read/write offset as well as I/O size distributions (NFS attribute count). For comparisons we chose streams which operated with distinct I/O sizes. We associate each 4 KB offset boundary as a separate bucket for all the 4 metrics. We considered buckets up to 100 MB, but in Figure 4(a) due to space constraints in the graph we only illustrate read I/O size distribution RMS error for request sizes of up to 1 MB. The RMS errors were bounded by 8%. More importantly, the I/O sizes not occurring in the original trace were never regenerated by the *Load DynamiX* streams. The same holds true for rest of the metric results as well. Results were bounded by similar error rates for the other distribution buckets as well. The reason for choosing 4 KB aligned distribution boundaries is that irrespective of whether a read is for a full block or for a few bytes within a block, our NFS server always reads a full block.

ParaSwift's success in Figures 3(b) and 4(a) is due to least error distribution fitting during the model building phase and the efficient translation of the corresponding function into an appropriate *Random* function parameters for *Load DynamiX*. ParaSwift's sampling technique carefully designs the BF functions to retain any variability in I/O size and file access offsets while sampling. Competitive RMS error values of the respective parameters for each stream against the 50% sampling, establishes the efficiency of our BF design.

**D)** Figure 4(b), illustrates the CDF of the file access popularity per a chosen stream of each workload. RMS errors of the total number of unique files per every 5 seconds interval were limited to 5% for all the sampled trace streams. Figure 4(b) illustrates the CDF for the top 25

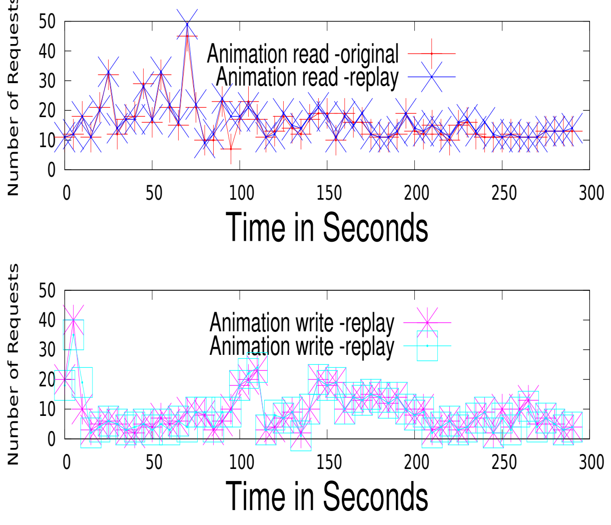


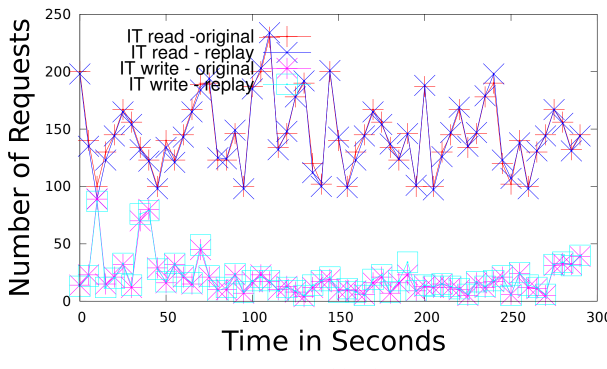
Figure 6: Animation Workload: Read and Write Requests, Original and Regenerated over a 5 minutes interval

files in the working set for a randomly chosen interval in the trace. We plotted the actual CDF vs. the replay CDF and found the errors to be less than 6%. Emulation of exact working set is accomplished due to recording of active file handles per operation of the adjacency matrix in the fileHandleMap data structure illustrated in Figure 2.

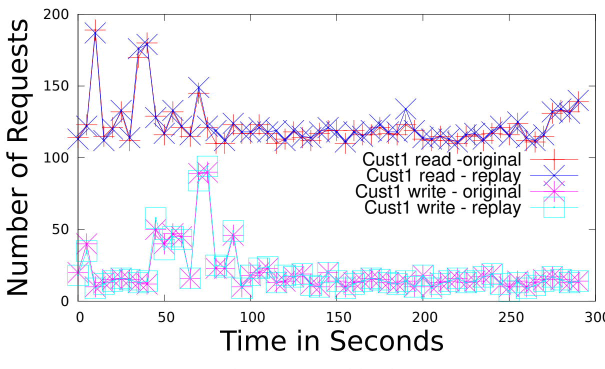
**E)** Figures 5(a), 5(b) and 6 represent comparison of the number of read/write requests generated for every 5 seconds interval by the most intense streams per workload trace. In reality, every new request issued by the *Load DynamiX* appliance is synchronous and response driven. However, the traces used by ParaSwift are from real deployments where the response is driven by the actual state of file system buffers, on disk fragmentation, and many other factors. In this experiment we want to assert the capability of our NFS client to generate the same intensity of read/writes when the various conditions pertaining to FS image and its state are identical. Hence, we turned off synchronous I/O mode and replay error notification in *Load DynamiX*.

We observe that the replay request rate closely follows the original rate for both read and write operations as the RMS errors were lower than 8%. This is attributed to ParaSwift's detailed calibration of load profiles in the form of 1) total number of workload patterns and 2) number of concurrent streams per workload pattern.

As mentioned earlier, these results hold true for other streams in the respective traces as well as for the larger sampled set.



(a) IT workload



(b) Cust1 workload

Figure 5: Read and Write Requests, Original and Regenerated over a 5 minutes interval

### 3.3.2 Sampling and ParaSwift Performance

**F)** Figures 7(a), 7(b) and 8 show the error vs. processing times for the corresponding streams of each workload with varying sampling frequency. Due to space constraints, we represent the error in these cases as the maximum of the RMS errors for metrics in A, B, C, D discussed above.

These figures show a slight drop in the accuracy (less than 10%) for up to the sampling rates of 60% with at least 5X boost in the processing speeds for all of the workloads. Beyond 60% we found the errors to be unbounded. ParaSwift can operate in two sampling modes: conservative and progressive. In the conservative mode, the actual sample in size may be more than expected as the requests could not be sampled as they were found to be unique by the BF. In the progressive mode, if the number of outstanding requests to be sampled exceeds a threshold, we discard the request irrespective of its novelty till the outstanding bar is pushed below the threshold. The results above come from the progressive mode. However, based on the physical characteristics of the trace, progressive sampling may degrade to a random sampling over time.

The same file handle can be accessed by multiple workload patterns simultaneously. Replay errors most likely result from such situations. The number of such situations cannot be either theoretically estimated or proved experimentally. Hence, we try to avoid such errors by advising *Load DynamiX* to re-create a new file of the same name if it does not exist in the FS image (at the time of its access). This is an inline replay correction method that we embed during the translation of the ParaSwift workload model to a *Load DynamiX* replayable project.

Finally, all the experiments reported in this paper were repeated thrice and the average reading was considered. Every replay was done with all the concurrent streams

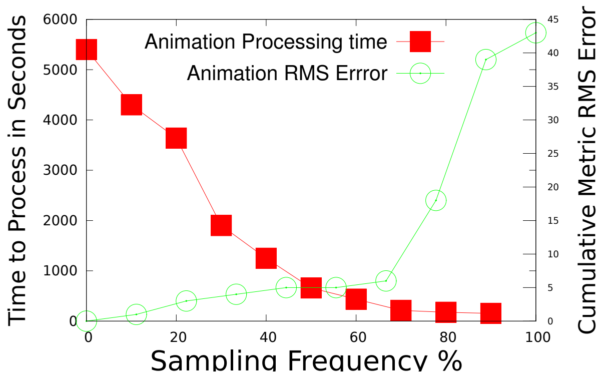
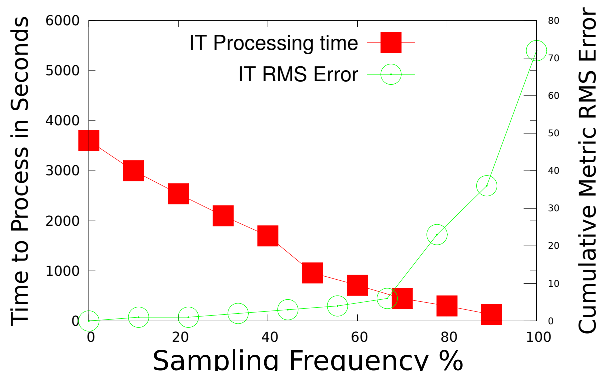


Figure 8: Animation Workload: RMS Error Vs. Compute times for the most intense streams with varying VM memory sizes

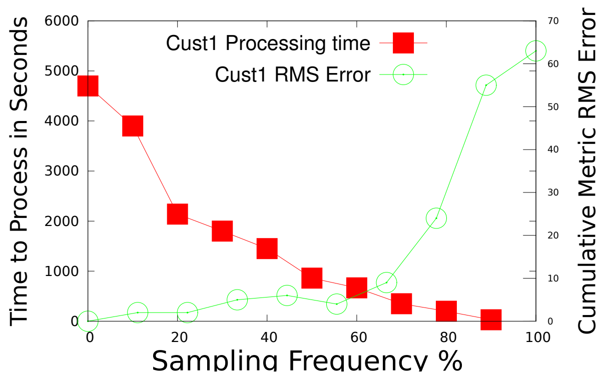
seen in the original trace run simultaneously as captured by ParaSwift load profile extractor in a dynamic way. Besides the RMS errors, the maximum errors in all of the above experiments were well below 12%.

## 4 Future Work

Statistically significant regeneration of multi-dimensional workload characteristics for NFS and SAN workloads is what we have primarily achieved via ParaSwift. However there are other aspects of workload regeneration essential for reproducing specific performance issue with trace replay. File system fragmentation is an important part and parcel of this problem. Another aspect deals with NFS file locking. While each of these problems are important by themselves they need a focussed investigation. ParaSwift is presently being used in our organization for reproducing customer problems dealing with workload characteristics alone, to validate various caching optimizations for various workload



(a) IT workload



(b) Cust1 workload

Figure 7: RMS Error Vs. Compute times for the most intense streams with varying VM memory sizes

verticals and stress testing our systems with specific workload segments. Appropriate aging of the file system and advanced features associated with stateful versions of NFS (V4) are a part of our future investigations.

## 5 Conclusion

In this paper we presented an algorithm/tool called ParaSwift that creates workload models based on traces, and then helps to replay them. This work addresses many of the open problems in the trace modeling and replay arena. ParaSwift implements a novel Bloom Filter based inline trace sampling mechanism that helps to both reduce the model creation time and also reduce the amount of memory that is consumed during the model building process. Our intention is to open source this code and enable the research community to build and distribute a large library of trace models that can be used to test future systems. Look out for the package ParaSwift on the world wide web in the near future. Finally, we have tested our end to end workflow using many real life traces and our results show that our techniques in general lead to less than 10% error in the accuracy of the trace model.

## References

[1] https://sites.google.com/site/murmurhash/.

[2] Load dynamix, inc. http://www.loaddynamix.com/.

[3] E. Anderson. Capture, conversion, and analysis of an intense nfs workload. In *Proccedings of the 7th Conference on File and Storage Technologies, FAST '09*.

[4] E. Anderson, M. Arlitt, C. Morrey, and A. Veitch. Dataseries: an efficient, flexible, data format for structured serial data.

[5] Peter B, Armando F, Michael J. F, Michael I. J, and D Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*.

[6] D. Christina, S. Sriram, K. Badriddine, V. Kushagra, and K. Christos. Time and cost-efficient modeling and generation of large-scale tpcc/tpce/tpch workloads. In *Proceedings of the Third TPC Technology conference on Topics in Performance Evaluation, Measurement and Characterization*.

[7] G. Ganger. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the Computer Measurement Group (CMG) Conference, 1995*.

[8] Maria E. Gomez and Vicente Santonja. Self-similarity in i/o workload: Analysis and modeling. In *Proceedings of the Workload Characterization: Methodology and Case Studies*.

[9] F. Hao, M. Kodialam, and T. V. Lakshman. Building high accuracy bloom filters using partitioned hashing. *SIGMETRICS Performance Evaluation Review, 2007*, 35 (1).

[10] M. Hazewinkel. Random function. Encyclopedia of mathematics, springer. 2001.

[11] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative i/o workloads using iterative distillation. In *11th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2003)*.

[12] SNIA. Iotta trace repository. `http://iotta.snia.org/tracetypes/3/`.

[13] R. Talwadker and K. Voruganti. Paragone: What's next in block i/o trace modeling. In *Proceedings of the Mass Storage Systems and Technologies, MSST '13*.

[14] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual machine workloads: The case for new benchmarks for nas. In *Proceedings of 11th USENIX conference on File and storage technologies, 2013*.

[15] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[16] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the USENIX ATC'12*.

# Pedagogical Framework and Network Laboratory Management Tool for Practical Network Administration Education

**Zahid Nawaz**
*University Of Oslo*
*Department Of Informatics, Oslo, Norway.*
glitterings@gmail.com

## Abstract

System Administration is an emerging academic discipline, yet there remains no commonly accepted approach to teaching the core skill of troubleshooting system problems. Troubleshooting is considered by many in the field to be the single most important skill for a working professional. The lack of common teaching methodology for this skill has resulted in an on the job training approach to teaching that is not scalable or measurable. This research addresses this gap with an approach that may be developed into a reliable, repeatable, scalable, and, most importantly, measurable approach to teaching the critical system administration troubleshooting skill. The research produced two core results, a proposed Pedagogical Framework to guide the teaching process, and a Network Laboratory Management Tool that is a proof of concept and basis for a more general purpose, practical implementation of the Framework. In this thesis the Framework and Tool are presented in detail and future work is described.

---

This is abstract of my Research Thesis in Master Programme "Network and System Administration " at University of Oslo, under Department of Informatics.

# An Investigation of Cultural and Organizational Impacts on EKR Use and EKR Use Outcomes Among System Administrators at NASA*

Nicole Forsgren
*Utah State University*
*3515 Old Main Hill*
*Logan, Utah 84322 U.S.A.*
nicolefv@usu.edu

E. Branson Matheson III
*Blackphone*
*rue du XXXI Décembre 47*
*CH-1207 Genève Switzerland*
branson@blackphone.ch

## Abstract

Organizations develop and implement Electronic Knowledge Repositories (EKRs) to support the codification, storage and reuse of knowledge. EKRs are seeing increased use in expert technical professions, such as system administration. Sysadmin.nasa.gov is such an EKR, initiated and supported by a grass-roots group of security and system administrator (sysadmin) professionals. The site supports several tools including a wiki, code repository, ticketing system, mailing lists and chat server to allow sysadmins to collaborate effectively across geographically distributed NASA centers.

Despite the increasing popularity of EKRs, an EKR is only as useful as the knowledge it holds. Referencing self-determination theory (Deci & Ryan 2000), the motivation-opportunity-ability framework (MacInnis et al. 1991), and the theory of knowledge reuse (Markus 2001), we include factors in our study that influence knowledge contribution to the EKR and knowledge sourcing from the EKR, as well as the impacts of EKR use on perceived work efficiency and effectiveness. Using a cross-sectional survey of 44 system administrators, we conducted our analysis of the use of an in-development EKR system in three steps, investigating (1) the effects of organizational culture and processes on sysadmin motivation to use the EKR; (2) the effects of motivation, costs, and technical factors on EKR use; and (3) the effects of EKR use on outcome measures, such as perceived performance. Overall the use of an EKR was found to be a net positive where, with proper support from management and consistent messaging, end-users were willing to share and use the tools and information. These findings are likely to generalize to similar organizational knowledge systems used by sysadmins.

*The effects of organizational culture and processes on sysadmin motivation to use the EKR:*

- If organizational rewards are associated with EKR use, messaging and support from management is key.
- Shared understanding of expertise and norms among sysadmins reduces the amount of satisfaction that comes from sharing knowledge through the EKR.
- The visibility of page ratings in EKRs contributes to the importance of extrinsic rewards.

*The effects of motivation, costs, and technical factors on EKR use:*

- While feeling time pressure reduces EKR use, sysadmins are still willing to spend time sharing knowledge with the system.
- Personal satisfaction with sharing knowledge contributes to EKR use; messages about sharing knowledge as a key aspect of being a sysadmin will increase EKR use.
- Consistent messaging and support from management is the most effective way to increase EKR use.
- System quality and usefulness positively impacts EKR use; any investments in EKR interface and search development will increase system use.

*The effects of EKR use on outcome measures, such as performance:*

- Sysadmins perceive efficiency and effectiveness gains from EKR use.
- The use of knowledge from external sources (such as web searches or vendor documentation) reflects negatively on the perceived value of early implementation EKRs. Continued support for EKRs is key in early rollout stages.
- Colleagues are seen as a complementary (not competing) knowledge source to EKRs.

[*] This work supported by NASA Langley Research Center Information Technology Enhanced Services (LITES) under Task Number A111.

## References

Deci, E.L., & Ryan, R.M. (2000). The "What" and "Why" of Goal Pursuits: Human Needs and the Self-Determination of Behavior, Psychological Inquiry, 11(4), 227-268.

MacInnis, D. J., Moorman, C., and Jaworski, B. J. 1991. "Enhancing and Measuring Consumers' Motivation, Opportunity, and Ability to Process Brand Information From Ads," Journal of Marketing (55:1), pp 32-53.

Markus, M. L. (2001). Toward a Theory of Knowledge Reuse: Types of Knowledge Reuse Situations and Factors in Reuse Success. Journal of Management Information Systems, 18(1), 57-93.

# Linux NFSv4.1 Performance Under a Microscope

Ming Chen,[1] Dean Hildebrand,[2] Geoff Kuenning,[3] Soujanya Shankaranarayana,[1]
mchen@cs.stonybrook.edu, dhildeb@us.ibm.com, geoff@cs.hmc.edu, soshankarana@cs.stonybrook.edu
Vasily Tarasov,[1,2] Arun O. Vasudevan,[1] Erez Zadok,[1] and Ksenia Zakirova[3]
{vass, aolappamanna, ezk}@cs.stonybrook.edu, kzakirova@g.hmc.edu
[1]*Stony Brook University,* [2]*IBM Research—Almaden, and* [3]*Harvey Mudd College*

NFS is a highly popular method of consolidating file resources. NFSv4.1, the latest version, has improvements in security, maintainability, and performance. We present a detail-oriented benchmarking study of NFSv4.1 to help system administrators understand its performance and take its advantage in production systems.

Our testbed consists of six identical Dell machines. One NFS server supports five clients via a 1GbE network. We began with a random read workload, where the five clients randomly read data from a 20GB NFS file for 5 minutes. In Linux 2.6.32, we observed that each client's throughput started at around 22MB/s but gradually decreased to around 5MB/s. We found the culprit to be the clients' read-ahead algorithm, which is aggressive and vulnerable to false-positive errors. The clients pre-fetched unneeded data and therefore wasted around 80% of the network's bandwidth. The read-ahead algorithm in Linux 3.12.0 is more conservative, and the clients achieved consistent 22MB/s throughput.

Switching to sequential reads, we observed a winner-loser phenomenon where three clients (winners) achieved a throughput of 28MB/s, while the other two (losers) got only 14MB/s. The winners and losers differed in multiple runs of the same experiments. This is caused by a HashCast effect on our NIC, which has eight transmit queues. The Linux TCP stack hashes each TCP flow to a particular NIC queue. The clients with collided hashes share one queue, and become losers because each queue has the same throughput. We note that multi-queue NICs are popular nowadays, and HashCast affects multi-queue-NIC servers hosting concurrent data-intensive TCP streams, such as file servers, video servers, etc.

We also benchmarked NFS delegation, which transfers control of a file from the server to clients. We found delegation especially helpful for file locking operations, which in addition to incurring multiple NFS messages, invalidate the locked file's entire client-side cache. Our micro-benchmark showed that NFS delegation saved up to 90% of network traffic and significantly boosted performance. Delegations are expected to benefit performance most of the time, since "file sharing is rarely concurrent". But it hurts performance if concurrent and conflicting file sharing does happen. We found that, in Linux, a delegation conflict incurs a delay of at least 100ms—more than $500\times$ the RTT of our network.

We found that writing NFS files with the `O_SYNC` flag, which causes more metadata to be written synchronously, has a side effect on the journaling of `ext4`, and can waste more than 50% of disk write bandwidth. We also noted that the TCP Nagle algorithm, which trades latency for bandwidth by coalescing multiple small packets, may hurt the performance of latency-sensitive NFS workloads. However, NFS in Linux has no mechanism to turn off the algorithm, even though the socket API supports this with the `SO_NODELAY` option.

By showing how unexpected behaviors in memory management, networking, and local file systems cause counterintuitive NFS performance, we call for system administrators' attention to NFSv4.1's intricate interactions with other OS subsystems. For a more flexible NFS, we urge the NFS developers to avoid hard-coded parameters and policies.

http://www.fsl.cs.stonybrook.edu/docs/nfs4perf/nfs4perf-microscope.pdf has more details.

# Demand-Provisioned Linux Containers for Private Network Access[*]

Patrick T. Cable II
*MIT Lincoln Laboratory*

## Abstract

System Administrators often need to have remote access to restricted networks that are separated for security reasons. The most common solution to this problem is to use a virtual private network (VPN) between the system administrator's client host to the restricted network. This solution exposes the restricted network directly to a potentially compromised client host. To avoid this direct network connection, an alternate solution is to configure an intermediate server, often called a bastion host, which serves as an explicit man-in-the-middle between untrusted and trusted networks. The bridge between networks is often established using secure shell (SSH). This solution reduces risk by implementing a central point of monitoring and ingress to the trusted network. Unfortunately, this also changes the bastion server's threat surface. Compromises to the intermediate server can result in the capture of authentication data (potentially from multiple users and for both the bastion itself or for assets on the private network) and can be a launch point for subsequent attacks.

To mitigate this risk, we have created an architecture that supports self-service provisioning of non-persistent bastion containers that are unique to each user. These containers only last for the duration of the connection, are only created after the client has authenticated with multiple factors, and perform live auditing outside the container to log user behavior in the private network. The system has four primary internal components: 1) a web based front end where users can request a session. 2) a controller on the compute host that manages the creation and destruction of containers, 3) the individual bastion containers, and 4) audit capabilities for both container creation and user monitoring inside of the containers.

This poster describes the system architecture and how we apply least privilege to each internal component to minimize risk. We also describe the implementation of our system, present initial results of its performance overhead, and walk through how a user would initiate a session.

[*]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. Approved for Public Release; distribution is unlimited.

# Virtual Interfaces for Exploration of Heterogeneous & Cloud Computing Architectures

Hal Martin
*University of Maryland Baltimore County*
*1000 Hilltop Circle*
*Baltimore, MD 21250*
hmarti2@umbc.edu

Wayne G. Lutters
*University of Maryland Baltimore County*
*1000 Hilltop Circle*
*Baltimore, MD 21250*
lutters@umbc.edu

## Abstract

In recent years data center operations have evolved into extensive infrastructures that can support of a wide range of computing paradigms, from traditional application hosting and data storage to service oriented architectures (SOAs) and emerging cloud services. Offering various mixes of software, platform, and infrastructure as a service (SaaS, PaaS, IaaS), more recent advances in software defined networking (SDN), combined with ubiquitous computing and IP convergence (voice, video, data) on end-point devices such as smart phones and tablets, have only added complexity to the delivery of business services.

This situation is further complicated by recent waves of mergers and acquisitions of these services by competing firms. In the commercial sector this has resulted in the creation of hybrid infrastructure resulting from the combination of the many different post-merger sites. These conglomerations are a soup of disparate applications, operating systems, data storage, communications protocols and networking fabrics, with various service and maintenance arrangements.The result is that information technology departments are are tasked with satisfying an ever expanding set of requirements and diversifying technical base.

One of the most daunting tasks post aggregation is the initial discovery and remote evaluation of the newly acquired, unexplored, legacy data center and network computing resources by an IT staff. When faced with performing discovery tasks, IT staff can be highly constrained by factors such as distance, time, risk identification, and knowledge from both a technology and corporate IT history standpoint. Research is required to understand the best practices for initial reconnaissance, combined with ongoing monitoring and analytics of the newly integrated infrastructure. This requires innovation in both system administration methods and tools.

While there have been some recent commercial advances, these IT vendor tool suites are expensive, complicated, and require significant resources to deploy. Their codebase and underlying methods remain proprietary. To advance the scientific understanding of these practices, we require assemblages of open source software tools and careful evaluation of human actors.

Thus, this project intends to advance the discipline by two major advances. The first is developing an instrumented evaluation testbed that provides generic infrastructure services, general user activity, and advanced computing constructs (Cloud, Software Defined Networking, etc.) in a simulated data center environment (SDCE). The second part is a Virtual, Interactive, Collaboration and Information Exchange Environment (VICkIEE), for performing such evaluations. Combined, these two components can be used for validating various data center configurations, evaluation methodologies, and tool suites for use in this task.

The VICkIEE is intended to be a real time, multi-tool, windowed UI with a shared collaborative interface to support multiple simultaneous system analysts. By remotely deploying the VICkIEE into a data center environment, operations can be performed with the same fidelity that local access provides. Future work intends to pursue user evaluations of the VICkIEE software suite to support discovery using a prototype SDCE in Fall 2014 with students enrolled in local cybersecurity and Information Technology programs.

# Time-Synchronized Visualization of Arbitrary Data Streams for Real-Time Monitoring and Historical Analysis[*]

Paul Z. Kolano
*NASA Advanced Supercomputing Division*
*NASA Ames Research Center, M/S 258-6*
*Moffett Field, CA 94035 U.S.A.*
paul.kolano@nasa.gov

## Abstract

Large installations involve huge numbers of interacting components that are subject to a multitude of hardware failures, transient errors, software bugs, and misconfiguration. Monitoring the health, utilization, security, and/or configuration of such installations is a challenging task. While various frameworks are available to assist with these tasks at a high level, administrators must more often than not revert to using command line tools on individual systems to get a low-level understanding of system behavior. The output from such tools can be difficult to grasp on even a single system, so when taken across a large number of hosts, can become completely overwhelming.

A variety visualization tools and techniques have been proposed to increase the amount of information that can be processed by humans at once. Existing tools, however, do not provide the flexibility, scalability, or usability needed to assist with all the varied information streams possible in large installations. In particular, these tools often require data in a specific format and/or in a specific location with interfaces that have little relation to the underlying commands from which the data originates.

Savors is a new visualization framework for the Synchronization And Visualization Of aRbitrary Streams. The goal of Savors is to supercharge the command-line tools already used by administrators with powerful visualizations that help them understand the output much more rapidly and with far greater scalability across systems. Savors not only supports the output of existing commands, but does so in a manner consistent with those commands by combining the line-editing capabilities of vi, the rapid window manipulation of GNU screen, the power and compactness of perl expressions, and the elegance of Unix pipelines. Savors was designed to support *impromptu visualization*, where the user can simply feed in the commands they were already using to create alternate views with optional on-the-fly aggregation of information across many systems. In this way, visualization becomes part of the administrator's standard repertoire of monitoring and analysis techniques with no need for a priori aggregation of data at a centralized resource or conversion of the data into a predefined format.

Savors can show any number of data streams either consolidated in the same view or spread out across multiple views. In multi-data scenarios, data streams can be synchronized by time allowing even distributed data streams to be viewed in the same temporal context. In single-data multi-view scenarios, views are updated in lockstep fashion so they show the same data at the same time. Together with its integrated parallelization capabilities, this allows Savors to easily show meaningful results from across even very large installations.

Savors consists of three components: a console, some number of data servers, and some number of views. The console is responsible for user interaction, spawning data servers and views according to the given command pipelines, and controlling synchronization between data streams. The data servers are responsible for spawning and interacting with the commands that generate data, manipulating the data as specified, and sending the data to the console and views. Finally, the views are responsible for visualizing the data as specified on one or more displays.

Savors is open source and available for download at `http://savors.sf.net`.

[*]This work is supported by the NASA Advanced Supercomputing Division under Task Number ARC-013 (Contract NNA07CA29C) with Computer Sciences Corporation

# Formalising Configuration Languages
## Why is this important in practice?

Paul Anderson
*University of Edinburgh*
dcspaul@ed.ac.uk

Herry Herry
*University of Edinburgh*
h.herry@sms.ed.ac.uk

***Formalising configuration languages sounds rather academic - why is this important in practice?***
Almost all large installations - including critical systems such as air traffic control[4] - now rely very heavily on configuration tools, and their associated languages, to define and manage their infrastructure. This growth means that configuration errors are becoming more serious, and more prevalent: they were found to be the second major cause of service-level failures in one of Google's main services[3], and they have been responsible for serious outages at most of the main internet service providers.
Unlike modern programming languages, configuration languages are usually defined by a single implementation, with no formal specification of the language - ambiguities and implementation errors are much more likely, and it is hard to create alternative compilers or other tools.

***How does formalisation help with these problems?***
• A precise definition enables us to create multiple implementations which are truly compatible and "correct".
• It also allows other people to experiment with compatible language extensions.
• And to implement supporting tools such as IDEs, graphical tools, analysers ...
• We can formally prove properties of the configuration making it highly reliable.
• Crucially, the process of developing the semantics also gives us a deeper understanding of the language and highlights problems with the language design.

***What did we do?***
We used an approach known as *denotational semantics* to develop a formal specification for the core of the SmartFrog[1] configuration language. SmartFrog is a declarative configuration language which is comparatively well-defined and has a typical structure. The denotational approach provides a direct mapping from the statements of the language onto their "meaning" - i.e. the real, resulting configuration.

***What did this achieve?***
• We used the semantics to prove some critical properties of the language, such as the fact that the compiler always terminates.
• Two people independently created three different implementations of the compiler[2] using the semantics as a specification. These were automatically compared using auto-generated, and hand-crafted examples, and found to be highly compatible. Two of these were extended to be fully compatible with the production compiler.
• We identified real bugs in the production compiler which has been in use for many years.
• We identified reasons for some "awkward" features in the language and possible ways of avoiding these.

***What did we learn from this?***
• It is possible to develop a formal semantics for configuration languages, and this helps to alleviate many of the practical problems mentioned above, and makes it much easier to create clear and correct compilers.
• This process is much more natural for "declarative" languages and further strengthens the case for their use (this bears comparison with the growth in popularity of functional programming languages).
• A more careful approach to the design and evolution of production configuration languages is necessary to avoid deep problems being caused by apparently small language changes.

**References**

[1] SmartFrog
http://www.smartfrog.org

[2] Demonstration SmartFrog compilers
https://github.com/herry13/smartfrog-lang

[3] Luiz André Barroso and Urs Hölzle. 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines

[4] Vervloesem K., FOSDEM 2011
http://www.lwn.net/Articles/428207