Using Trātṛ to tame Adversarial Synchronization

Yuvraj Patel*, Chenhao Ye, Akshat Sinha, Abigail Matthews,
Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Michael Swift
University of Wisconsin-Madison

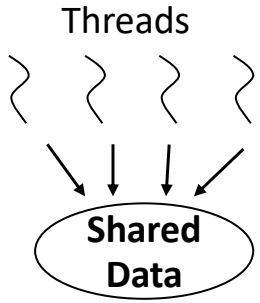
* Joining the University of Edinburgh as Assistant Professor(Lecturer as per UK parleys) in September'22

Modern Systems Are Concurrent

Harness concurrency to achieve high performance

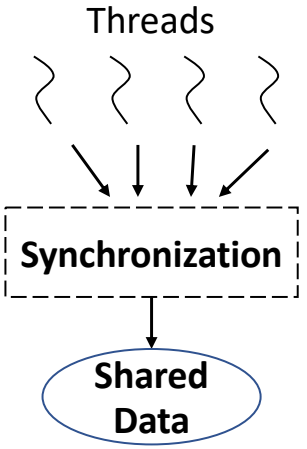
Correctness & Synchronization

Uncoordinated access



Correctness???

Coordinated access



Locks are widely used to ensure mutual exclusion

Linked List Example

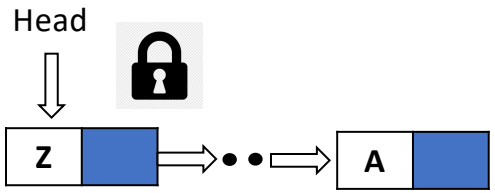
Linked list protected using a single lock

Insert() – Node added at the head

- Time spent holding the lock is tiny

Search() – Traverse list to find entry

- Time spent holding the lock may be longer depending on the list size



Cooperative Aspect Of Synchronization

Within an application, threads cooperate towards same goal

No misbehavior expected

Any impact of synchronization contained within application

- Change locks, refactor code or ignore

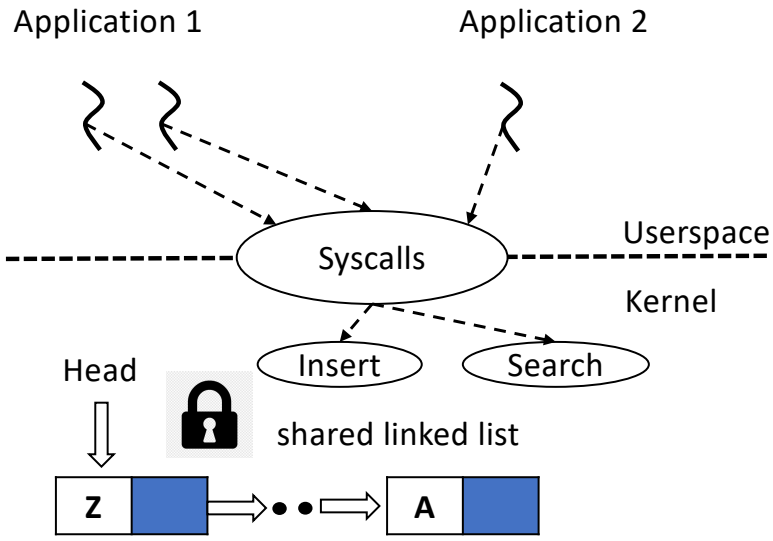
Sharing Across Applications

Locks used in competitive environments

Cooperation expected but applications may misbehave

Size of the list matters

- Billions of entries – Security issue
- Applications may observe poor performance and denial-of-services



The Problem – Adversarial Synchronization

Attack shared data structures and synchronization primitives

- Artificially grow data structures leading to lock contention
- Leads to poor performance and denial-of-service

Impacts multiple (or all) victims/tenants

Two types of attacks

- Synchronization attack – Attacker actively participates in lock acquisition
- Framing attack – Attacker turns passive after expanding data structure

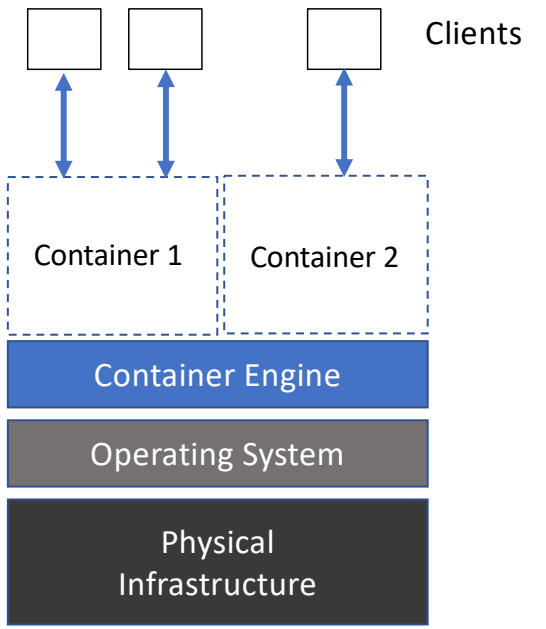
Threat Model

One or more containers run on a single physical machine

All containers run arbitrary, unprivileged workloads that access OS services

1-1 mapping between tenants and users

No collusion amongst malicious users



Real-world Attack

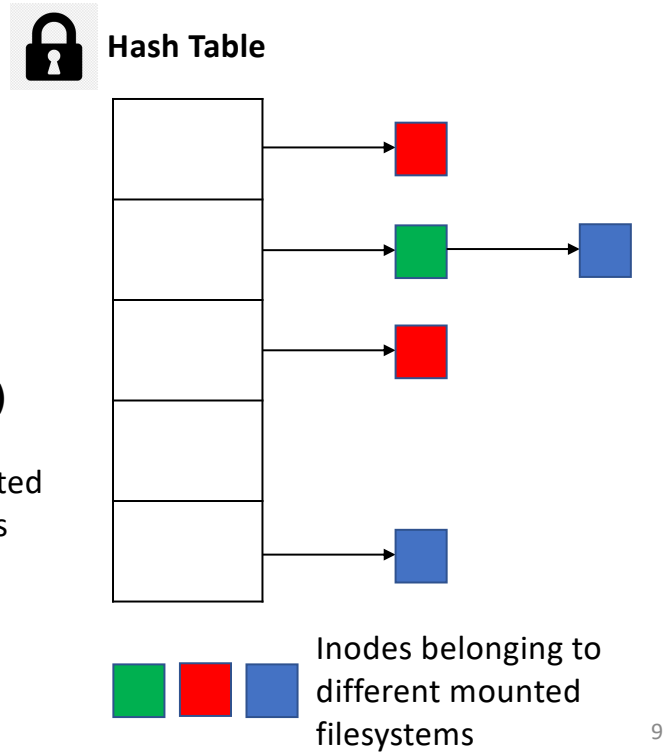
Inode cache @ Linux Kernel

- Caches inodes in memory
- Shared across all mounted filesystems
- Global lock protects the inode cache

hash = hash_function (Inode number, Superblock pointer)

Inode number – Unique number for each file

Superblock pointer – Memory address for mounted filesystem; Value varies across remounts/reboots



Real-world Attack

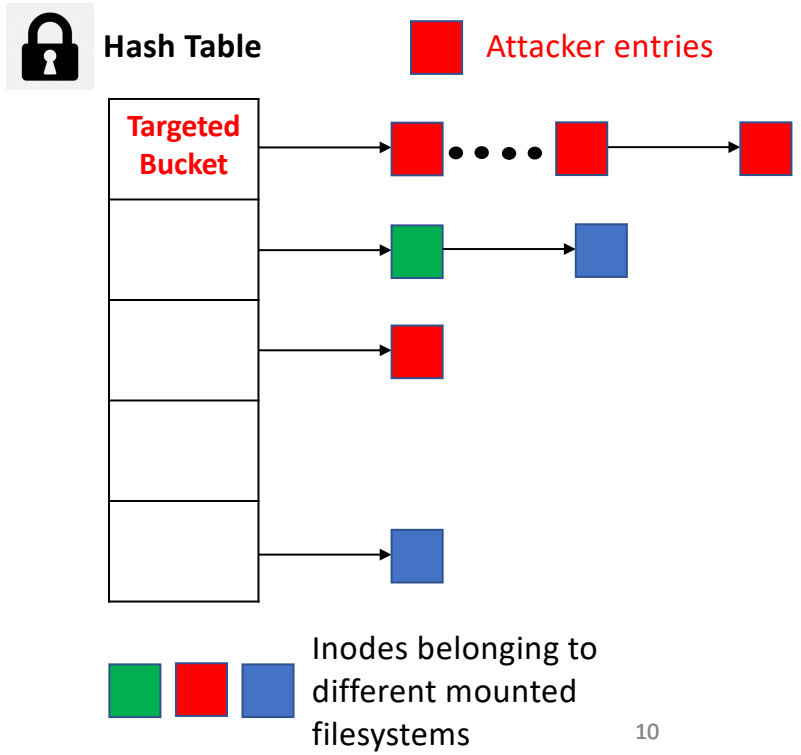
Managed to break hash function to identify superblock pointer

Attacker

- Mount FUSE user-space filesystem that lets one choose inode numbers
- Choose inode numbers that collide to identify superblock pointer
- Create many files targeting a hash bucket

Victim

- Waits while trying to access the inode cache

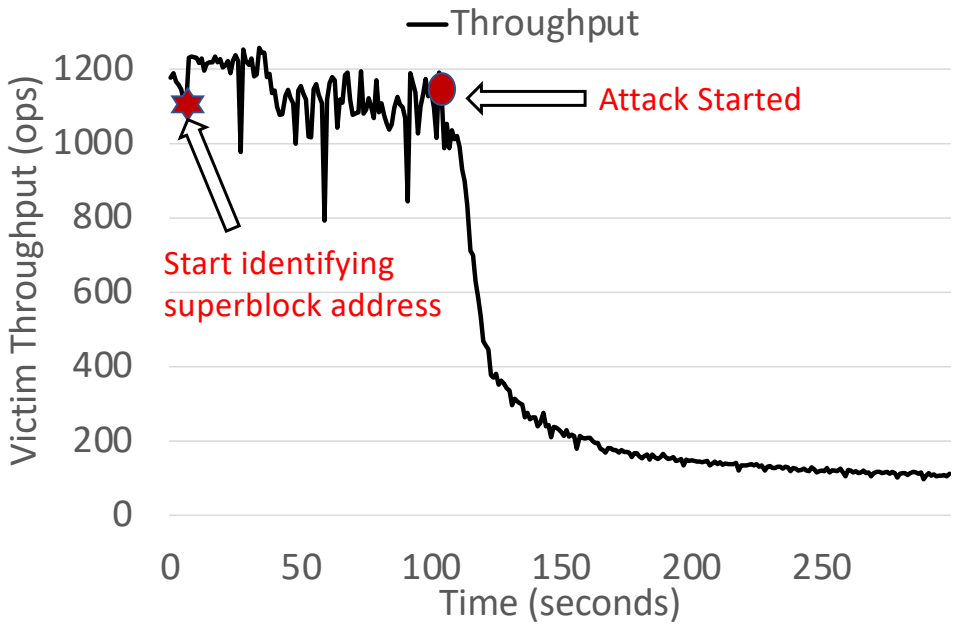


Inode Cache Attack

Experiment

- Exim Mail server running as victim container
- Accesses inode cache frequently

Throughput reduces by 92% (12X)

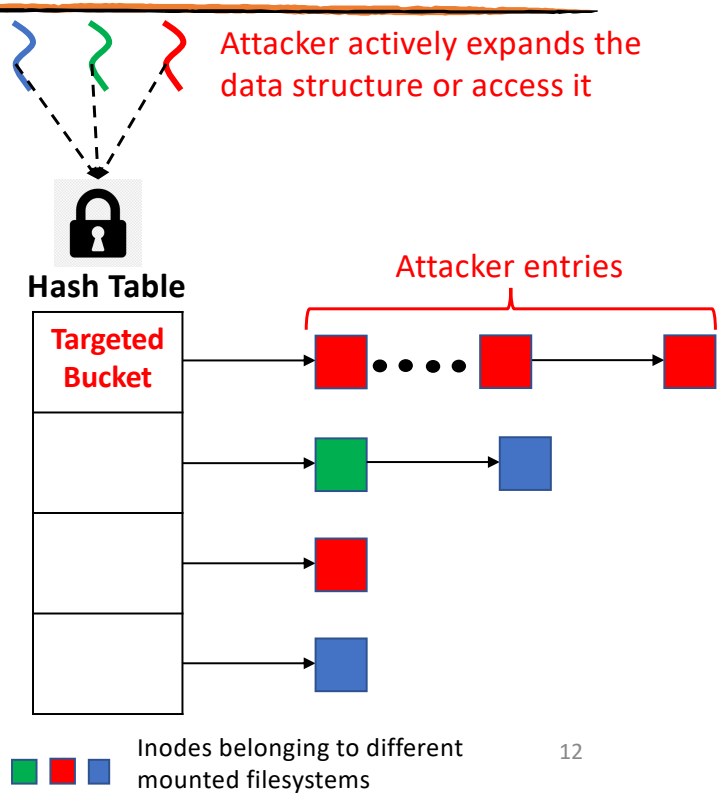


Synchronization Attacks

Attacker expands the data structure[1] and actively acquires lock repetitively

Attacker holds lock longer

Victims wait-time increases



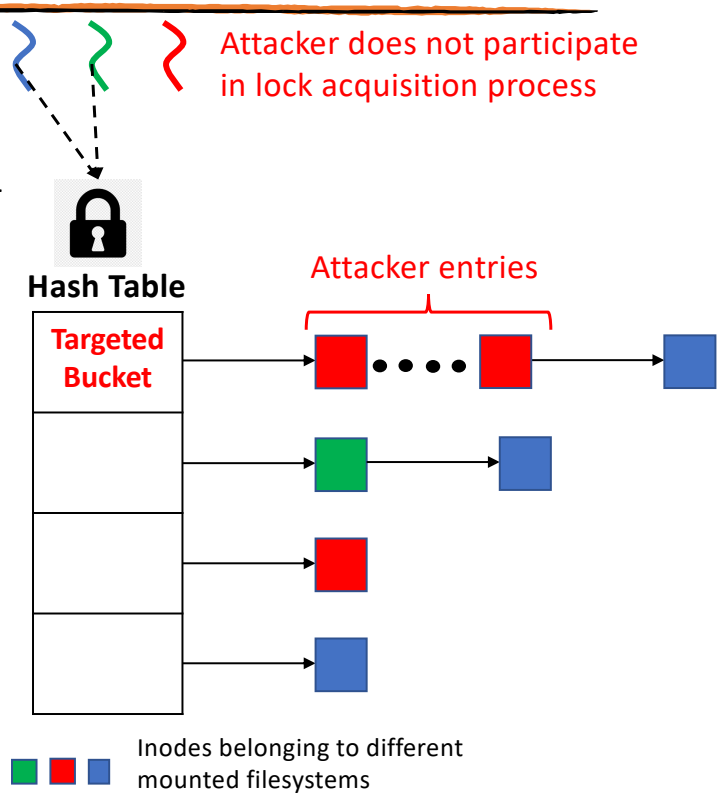
1. Data structures that exhibit weak complexity can trigger worst-case performance

Framing Attacks

Attacker expands the data structure and does not participate in lock acquisition process

Victims traverse the expanded data structure[1]

- More time in critical section/More CPU usage
- More time waiting to acquire the lock



1. Like framing in crime, inspection of who holds the lock longer will incorrectly frame innocent victims.

Trātṛ – Taming Adversarial Synchronization

Trātṛ – Linux kernel framework to detect and mitigate attacks

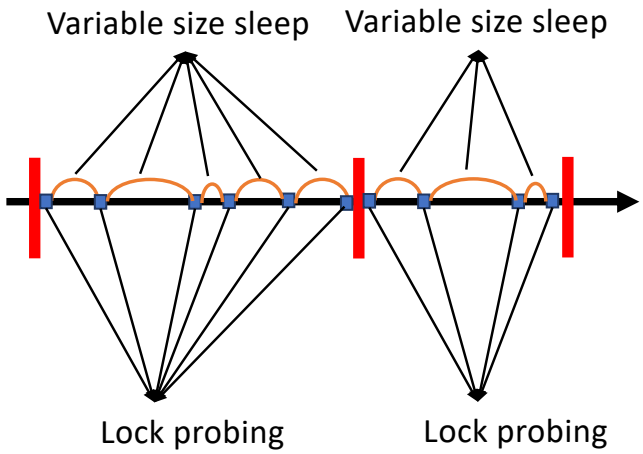
Incrementally add support for data structures

Four mechanisms – Tracking, Detection, Prevention, Recovery

Are We Under An Attack?

Detect attacks by probing lock wait-times

- Random checks
- Check wait-times more than set threshold
- Confirm multiple times to avoid false positives

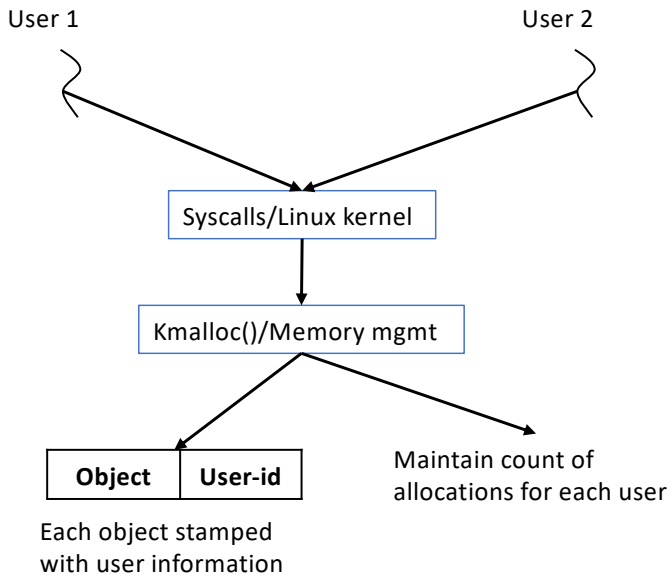


Who's The Culprit?

Identify an attacker by enabling tracking of kernel objects

Record kernel object allocations per user

Stamp tenant information on every object



Maintain information for every type of kernel object

How to Respond?

Two crucial aspects

- Stop future attacks and further worsening of the attack
- Revert to pre-attack state

Prevent future expansion of the data structure

- Stop attacker from allocating more objects

With prevention only

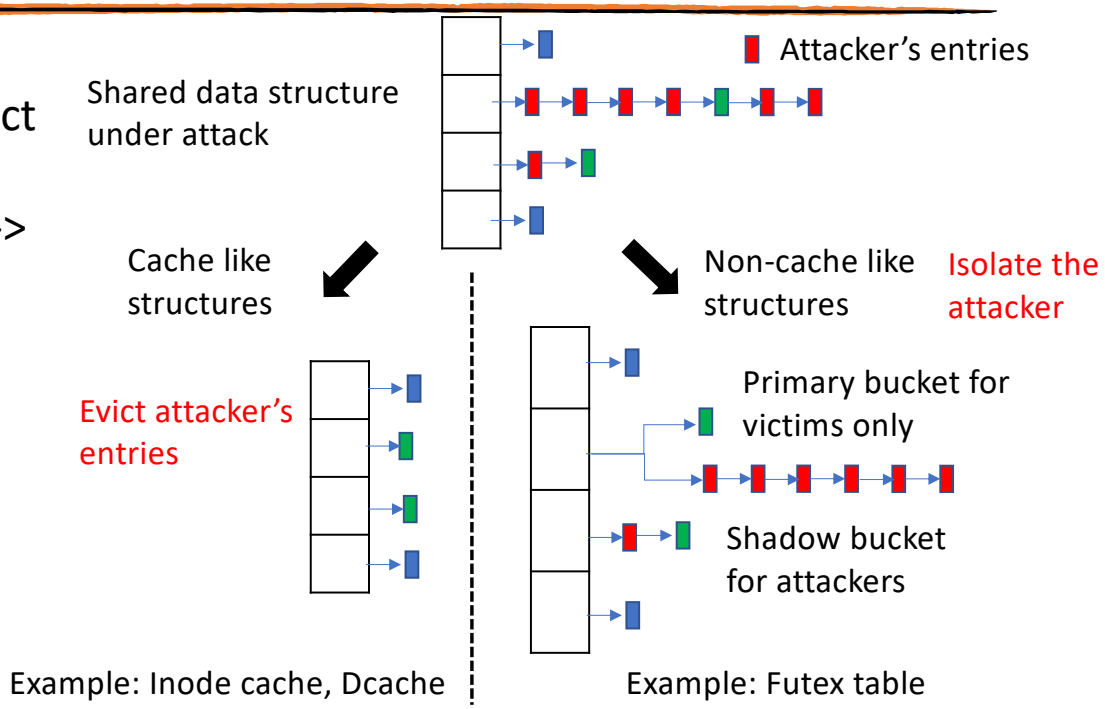
- Attacker can access the data structure and continue synchronization attack
- Framing attacks still possible

Undoing the damage crucial

How To Undo The Damage?

Different solutions

- **Cache like structures** -> Evict attacker entries
- **Non-cache like structures** -> Isolate attacker



Shared data structure after recovery

Trātṛ in Action – Inode Cache Attack

Superblock pointer known

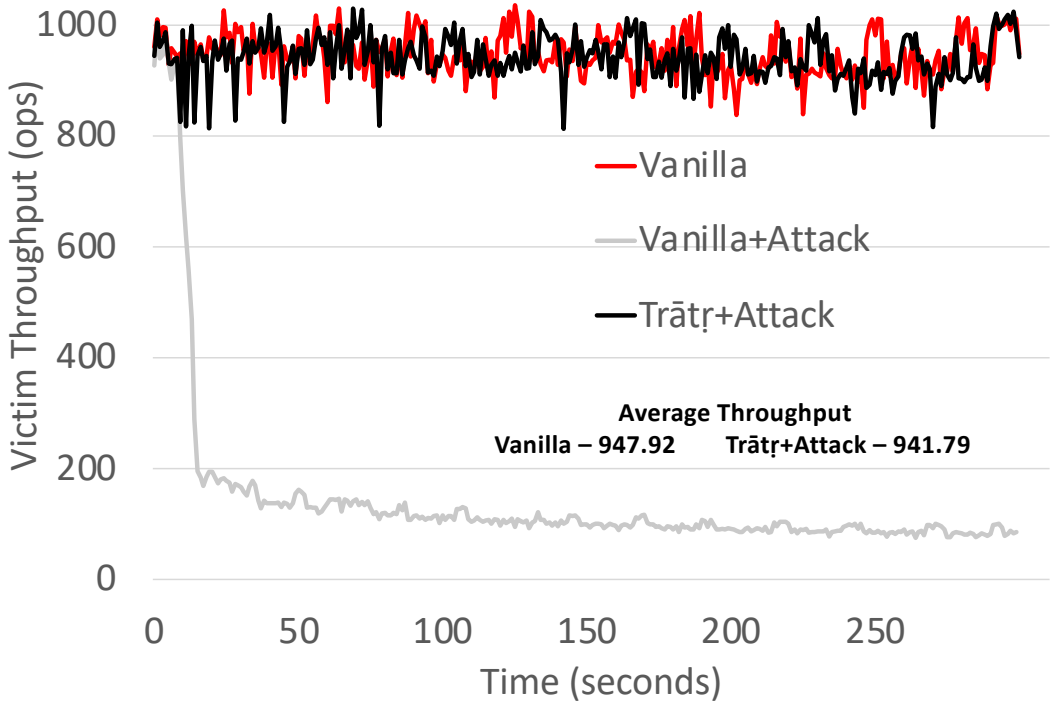
Compare native Linux & Trātṛ

Vanilla under attack

- Throughput reduces by 12X

Trātṛ – Quickly detect and mitigate attack

- Minimal performance impact



Experiment

Exim Mail server running as victim container

Attacker launches inode cache attack

Other Results Summary

Performance

- Responsive – Can detect attacks within seconds of launching an attack
- Recovery completes within 100's of milliseconds
- Overhead
  - Minimal memory overhead (0-20 MB increase in slab cache memory usage)
  - Minimal tracking overhead (0-4% performance reduction)
  - Minimal other mechanisms overhead (1.5% performance reduction)

Easy to add a new data structure (150 lines of new code)

False negative and positive study

Conclusion

Introduce a new problem – Adversarial Synchronization

- Attacker attacks synchronization primitives leading to poor performance and denial-of-services
- Two types of attacks – Synchronization and Framing attacks

Introduce Trātṛ to mitigate adversarial synchronization

- Linux-kernel framework that support incremental adding of support
- Evaluation shows Trātṛ is responsive, efficient, and effective
- Attack scripts and Trātṛ Source accessible at: https://research.cs.wisc.edu/adsl/Software/

Thank you for listening
Questions?