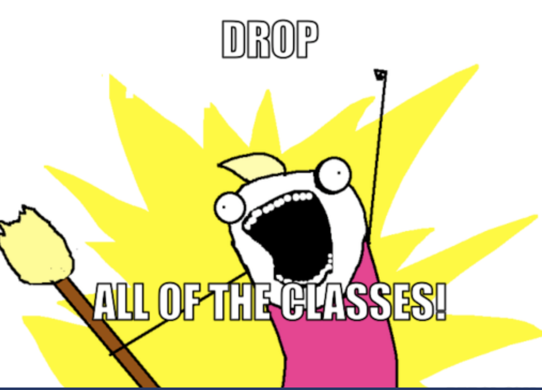
# CS 563 - Advanced Computer Security: System Intrusions

Professor Adam Bates
Fall 2018

# Administrative

**Learning Objectives**:
- …
- Survey broad topics in the "system intrusions" area

**Announcements**:
- Reaction paper was due today (and all classes)
- Feedback for reaction papers soon
- "Preference Proposal" Homework due 9/24 (next slide)
- 33 students left in the course as of yesterday
  - ~= 1 Paper presentation per student?

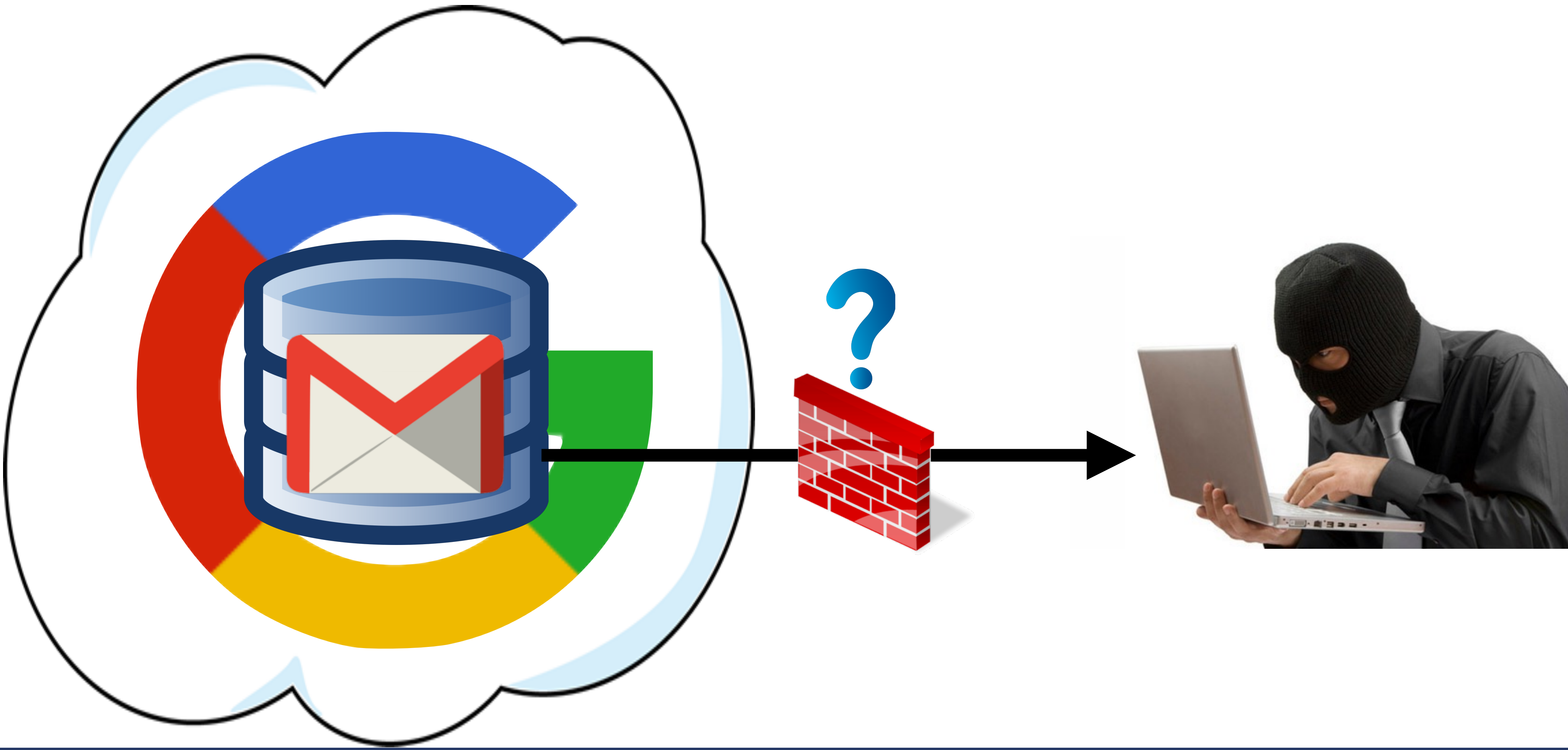



**Reminder**: Please put away (backlit) devices at the start of class

# System Intrusions

We live in an age of high profile data breaches…

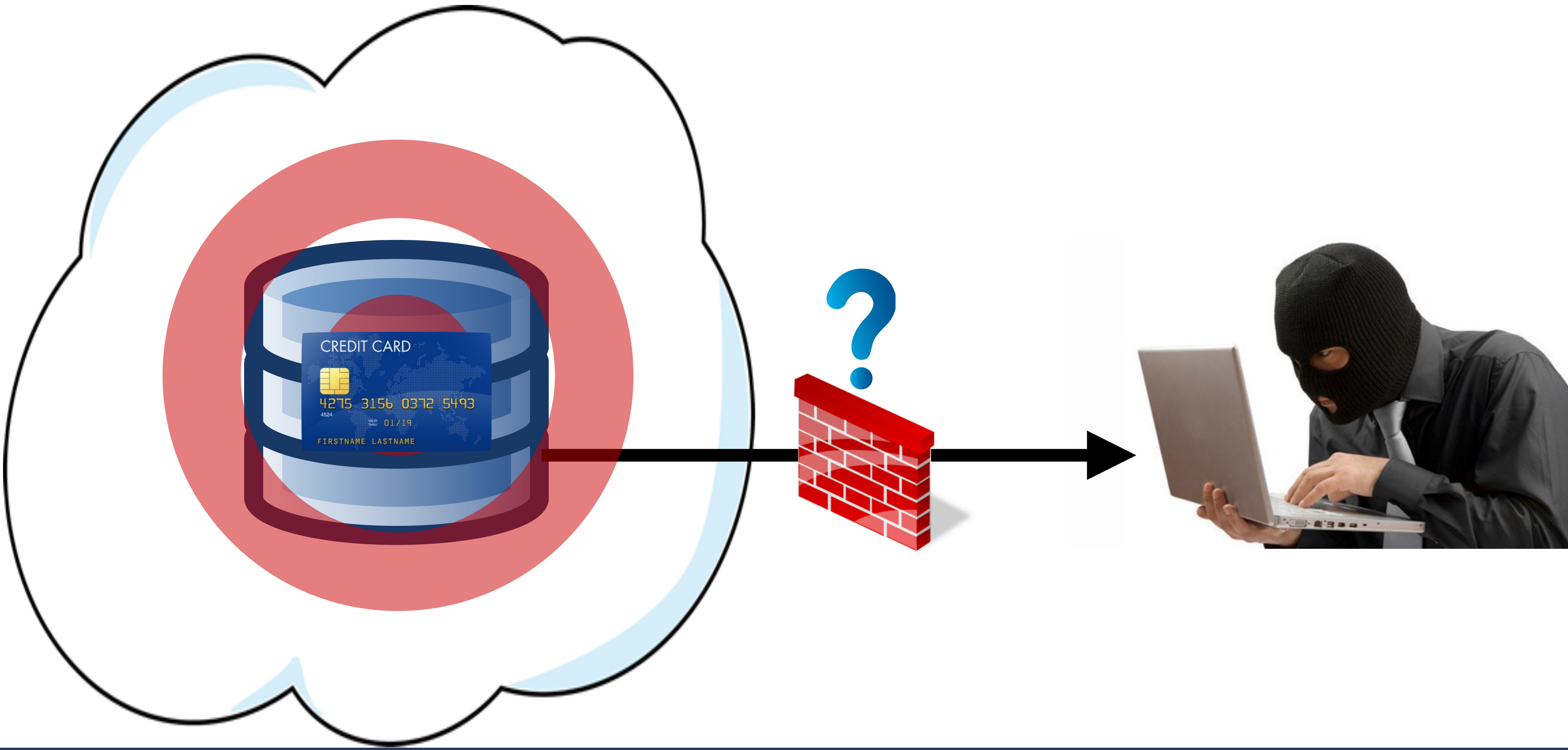
*<u>Operation Aurora</u>: Google Mail was subject to a sustained nation state attack for the entire year of 2009.*



# System Intrusions

We live in an age of high profile data breaches…

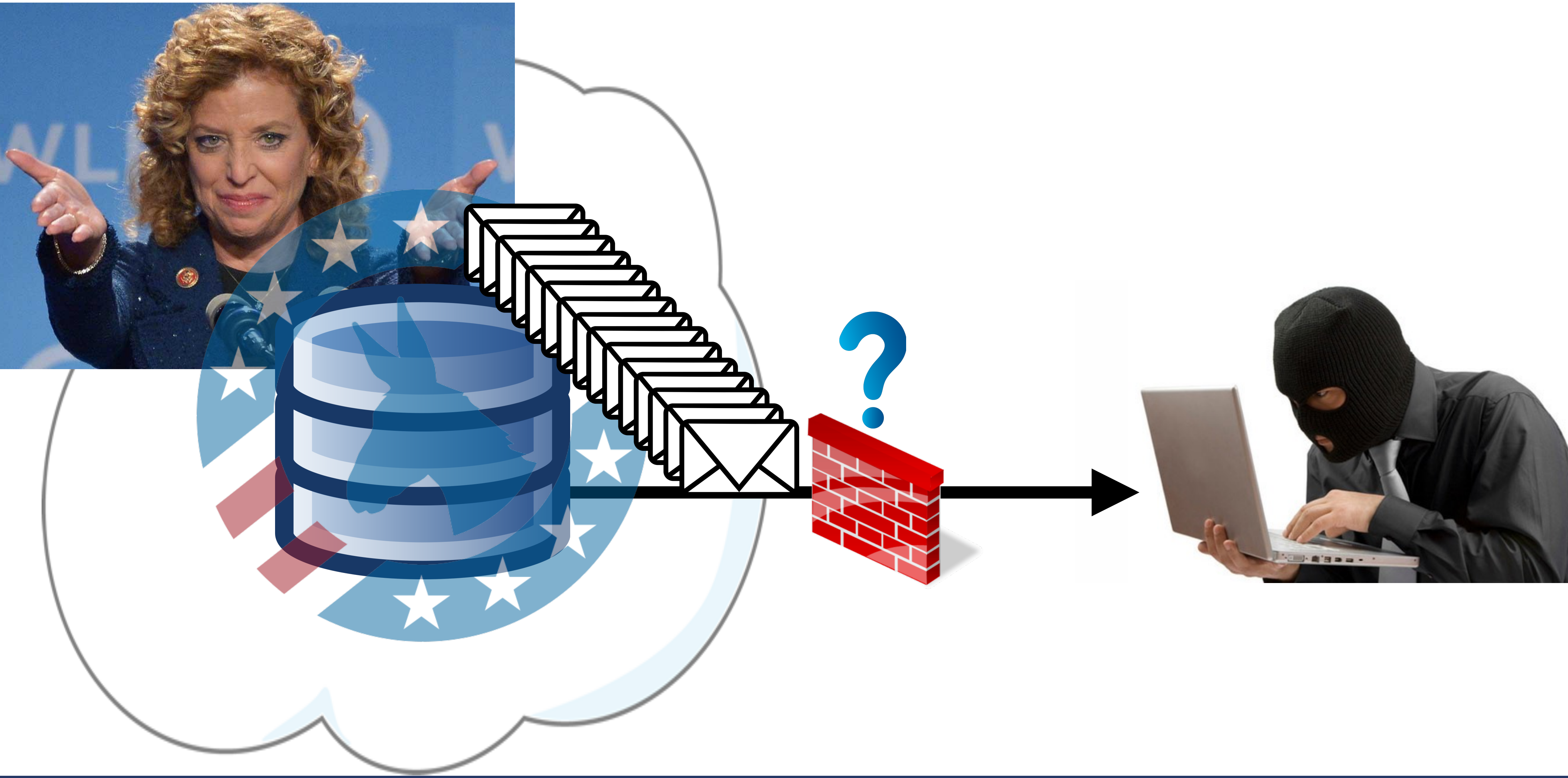
*Target loses 70 million credit card numbers…*



# System Intrusions

We live in an age of high profile data breaches…

*DNC loses 30 thousand emails…*



# System Intrusions

We live in an age of high profile data breaches…

Forbes

Security / #CyberSecurity

SEP 7, 2017 @ 10:42 PM 41,538

## Equifax Data Breach Impacts 143 Million Americans

Black Duck Blog

**Equifax, Apache Struts, & CVE-2017-5638 Vulnerability**

Written by Fred Bals | Senior Content Writer/Editor | Sep 15, 2017

**Equifax Inc.** @Equifax

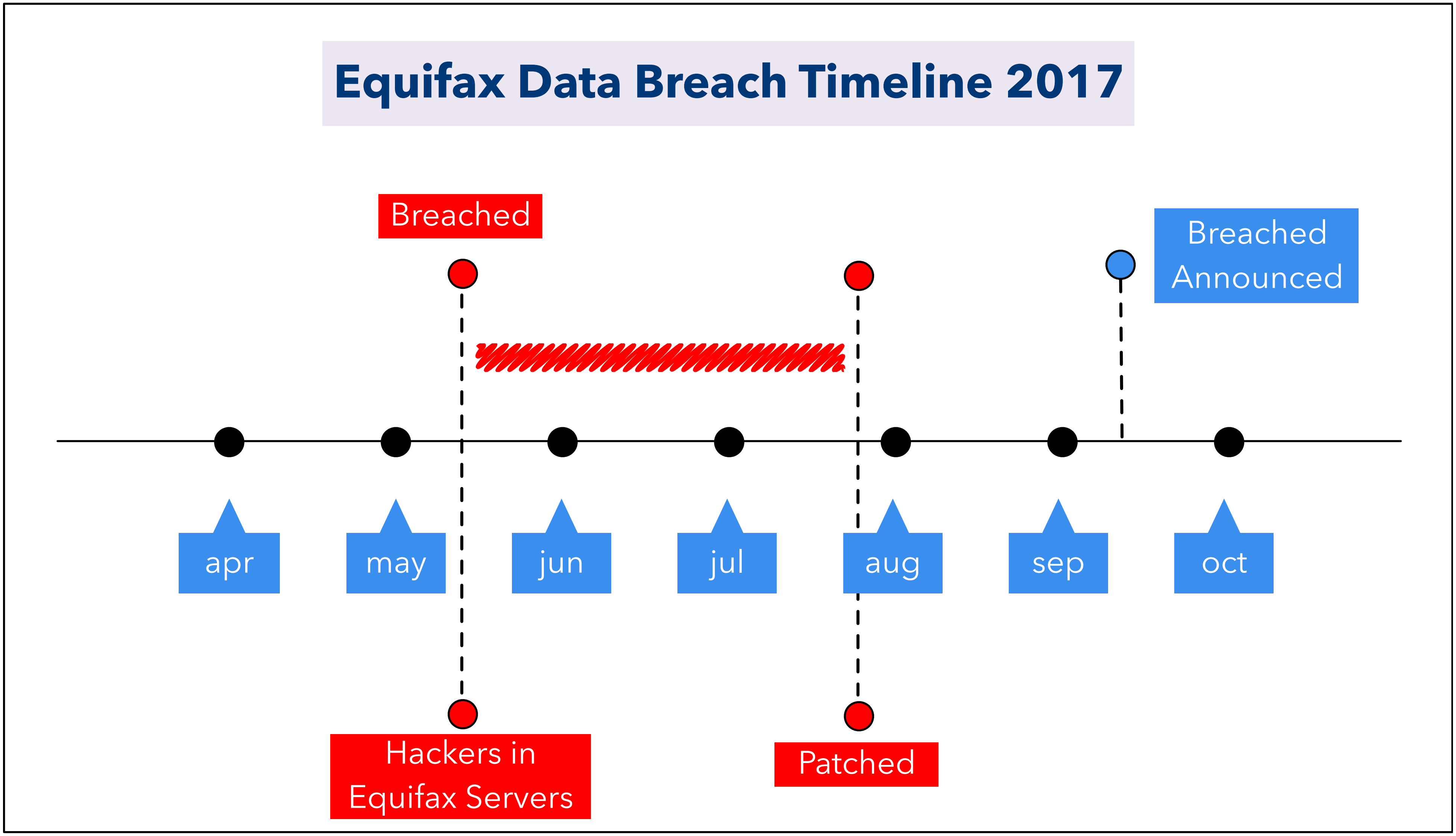
Follow

We recently discovered a cybersecurity incident involving consumer information. Once discovered, we acted immediately to stop the intrusion.

6:20 PM - 7 Sep 2017

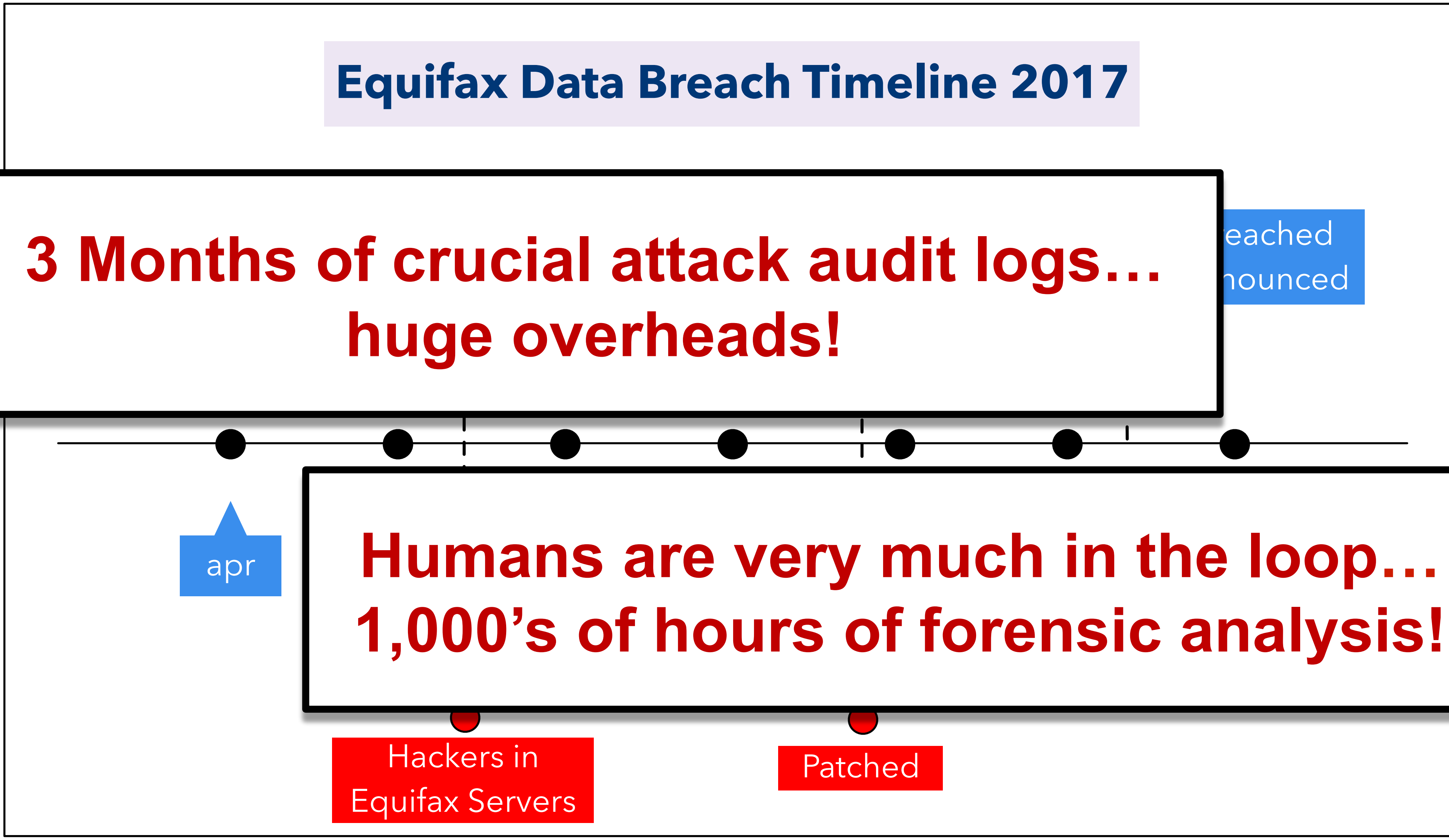
# System Intrusions

We live in an age of high profile data breaches…



# System Intrusions

We live in an age of high profile data breaches…




**3 Months of crucial attack audit logs…**
**huge overheads!**

**Humans are very much in the loop…**
**1,000's of hours of forensic analysis!**

# System Intrusions

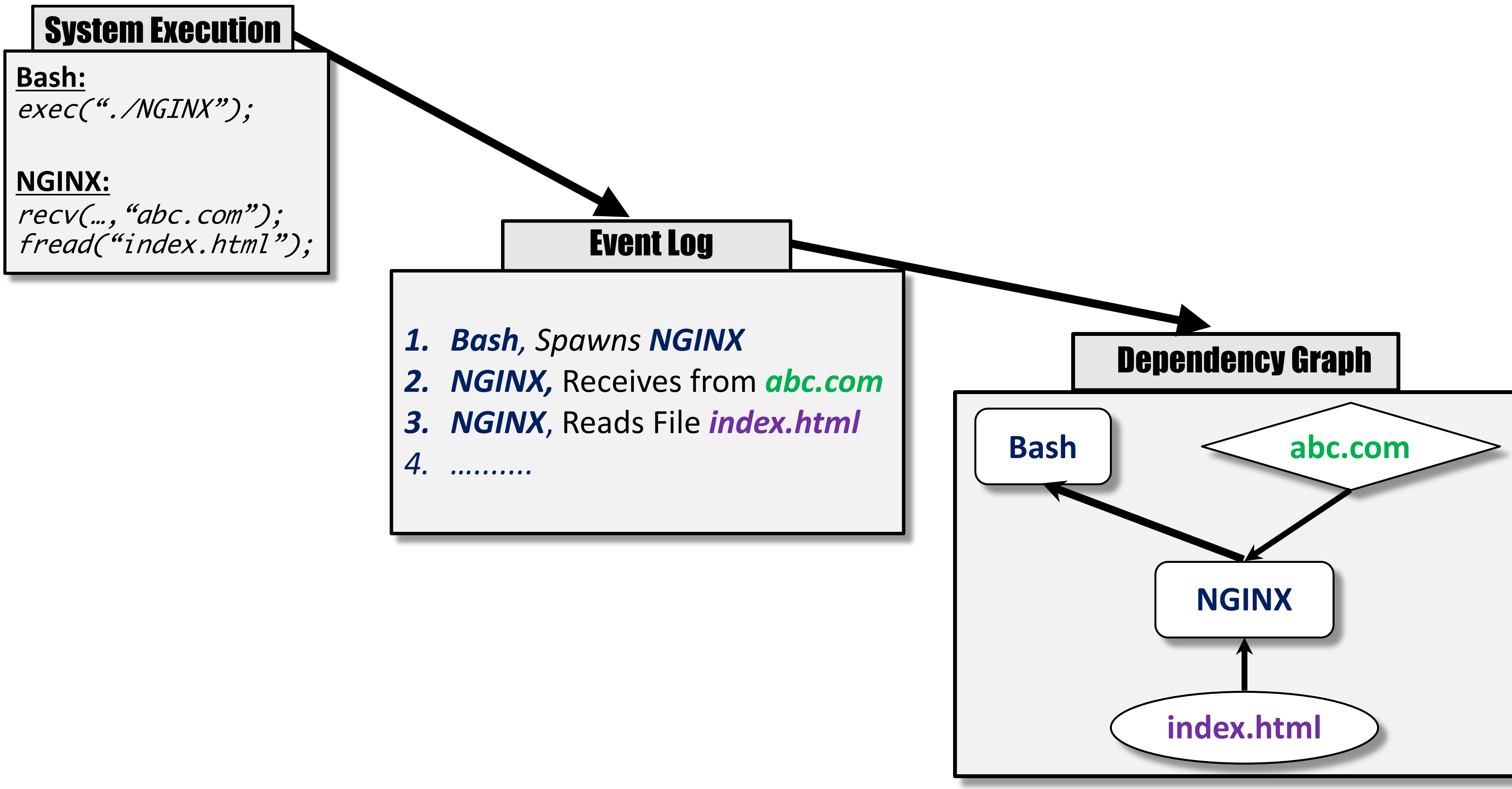
We live in an age of high profile data breaches…

**How can we make sense of the available forensic data?**

**Can we understand the attacker in time to prevent them from reaching their goal?**

# Backtracking Intrusions

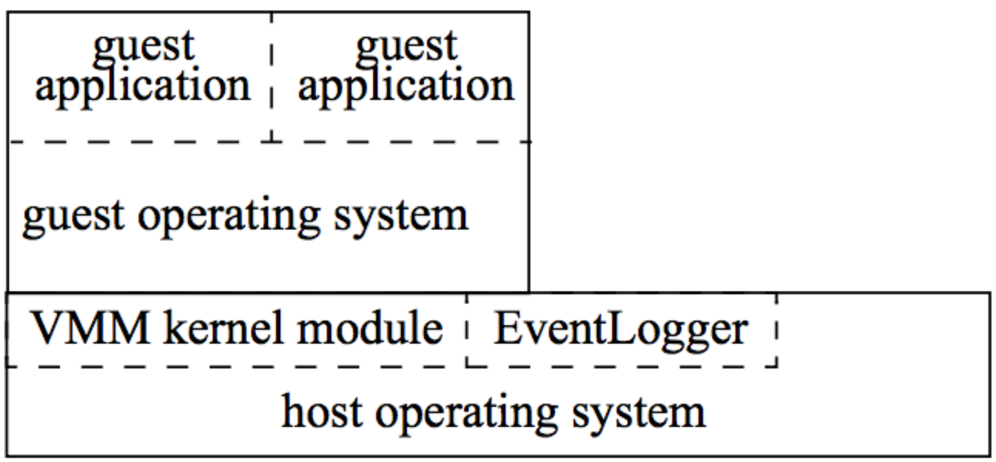
Idea: Parse individual system events into relationship graphs




[King and Chen, SOSP'03]

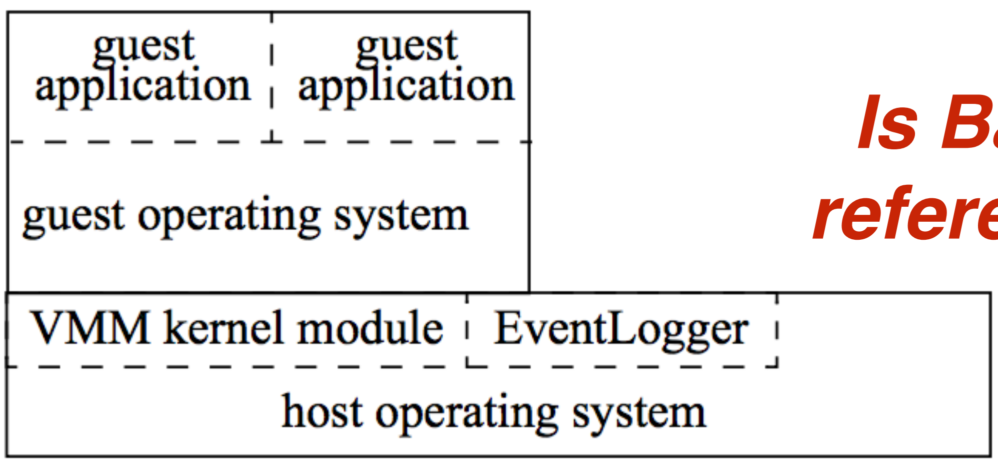
# BackTracker

- Observes OS-level events
  - Objects: processes, files, filenames
  - Traces System Call Events: Process/Process, Process/File, Process/Filename
  - ***Alternatives? Why OS level?***
- Constructs dependency graph offline
- Filters graph for more succinct explanations
- EventLogger mechanism embedded in virtual hypervisor hosting target system



# BackTracker

- Observes OS-level events
  - Objects: processes, files, filenames
  - Traces System Call Events: Process/Process, Process/File, Process/Filename
  - ***Alternatives? Why OS level?***
- Constructs dependency graph offline
- Filters graph for more succinct explanations
- EventLogger mechanism embedded in virtual hypervisor hosting target system



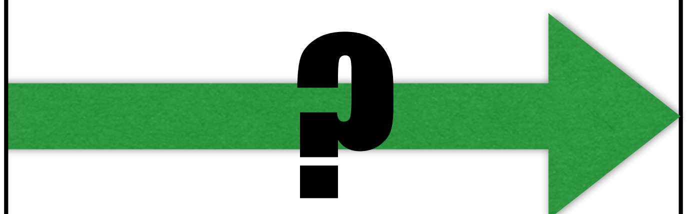

***Is BackTracker a reference monitor?***

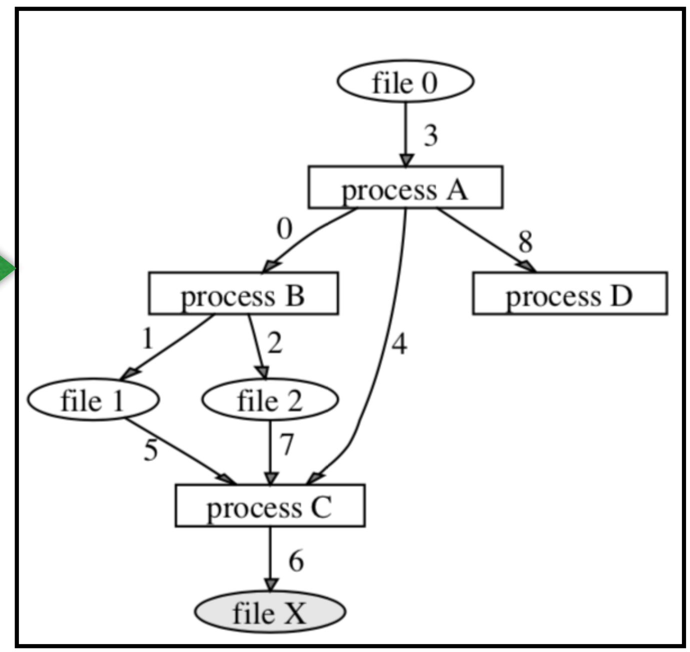
# Dependency Types

- <u>High-Control Events</u>: Events through which an attacker can directly "accomplish a task" (i.e., security-critical)
  - Ex: write or read a file, create a process
- <u>Low-Control Events</u>: Events through which an attacker might indirectly "accomplish a task" by affecting another process
  - Ex: modify file metadata, create directory entries
- BackTracker primarily supports tracking of high-control events.
- ***Thoughts on this?***

**Dependency graphs vs. backtraces….**

time 0: process A creates process B
time 1: process B writes file 1
time 2: process B writes file 2
time 3: process A reads file 0
time 4: process A creates process C
time 5: process C reads file 1
time 6: process C writes file X
time 7: process C reads file 2
time 8: process A creates process D



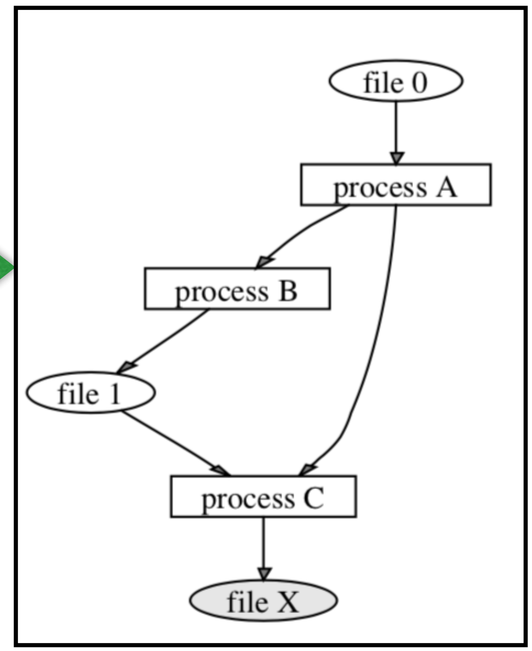
# Graph Construction

**Dependency graphs vs. backtraces….**

```
foreach event E in log { /* read events from latest to earliest */
    foreach object O in graph {
        if (E affects O by the time threshold for object O) {
            if (E's source object not already in graph) {
                add E's source object to graph
                set time threshold for E's source object to time of E
            }
            add edge from E's source object to E's sink object
        }
    }
}
```

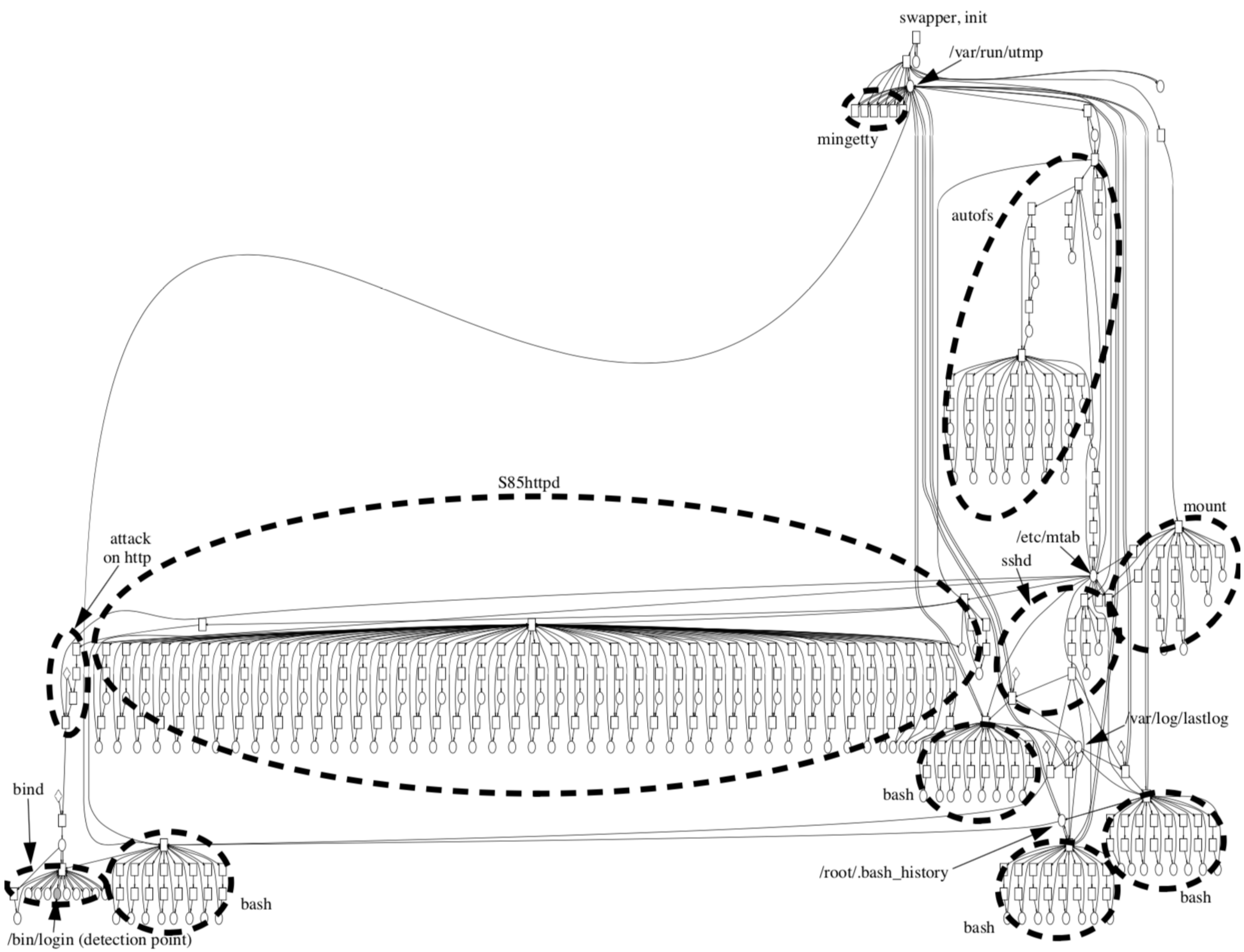
**Figure 2: Constructing a dependency graph.** This code shows the basic algorithm used to construct a dependency graph from a log of dependency-causing events with discrete times.

time 0: process A creates process B
time 1: process B writes file 1
time 2: process B writes file 2
time 3: process A reads file 0
time 4: process A creates process C
time 5: process C reads file 1
time 6: process C writes file X
time 7: process C reads file 2
time 8: process A creates process D



# Filtering

Even backtraces (i.e., dependency subgraphs) get real big, real fast…



# Filtering

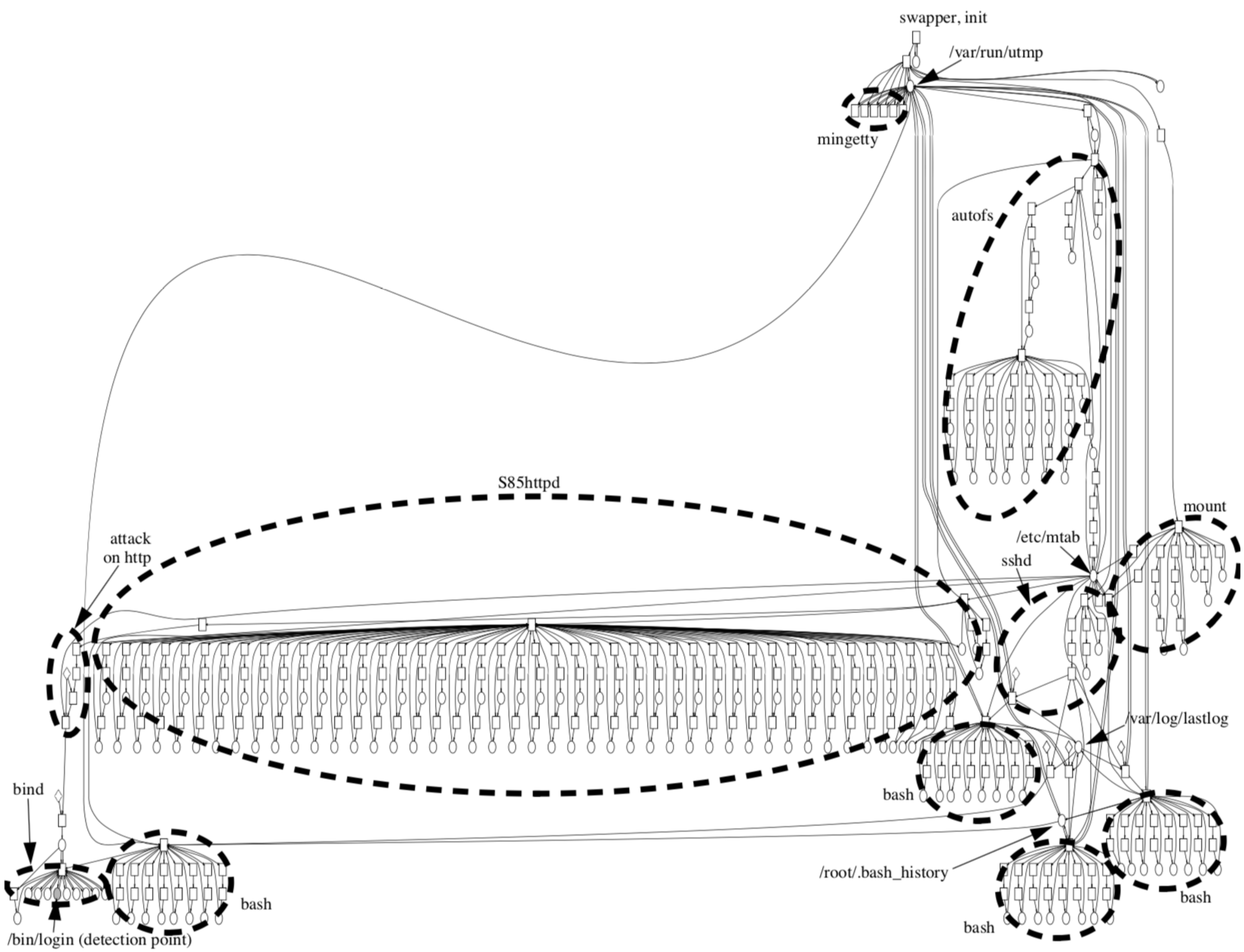
Even backtraces (i.e., dependency subgraphs) get real big, real fast…

Filtering Strategies

- Blacklist objects or event types
- Prune read-only files from graph
- Prune helper applications from graph (***how?***)
- Calculate the intersection of multiple detection points

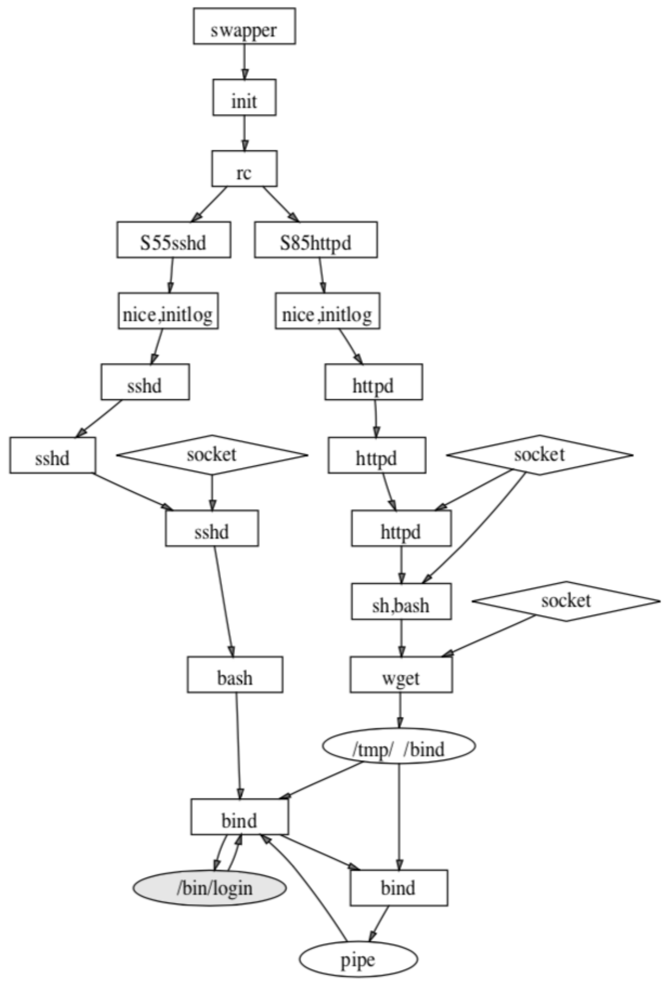
# Filtering

Even backtraces (i.e., dependency subgraphs) get real big, real fast…



# Filtering

Even backtraces (i.e., dependency subgraphs) get real big, real fast…



# Evaluation

Multiple real attacks against honeypot ReVirt VM, plus one synthetic attack…

| | *bind* (Fig 5-6) | *ptrace* (Fig 1) | *openssl-too* (Fig 7) | *self* (Fig 8) |
|---|---|---|---|---|
| time period being analyzed | 24 hours | | 61 hours | 24 hours |
| # of objects and events in log | 155,344 objects<br>1,204,166 events | | 77,334 objects<br>382,955 events | 2,187,963 objects<br>55,894,869 events |
| # of objects and events in unfiltered dependency graph | 5,281 objects<br>9,825 events | 552 objects<br>2,635 events | 495 objects<br>2,414 events | 717 objects<br>3,387 events |
| # of objects and events in filtered dependency graph | 24 objects<br>28 events | 20 objects<br>25 events | 28 objects<br>41 events | 56 (36) objects<br>81 (49) events |
| growth rate of EventLogger's log | 0.017 GB/day | | 0.002 GB/day | 1.2 GB/day |
| time overhead of EventLogger | 0% | | 0% | 9% |

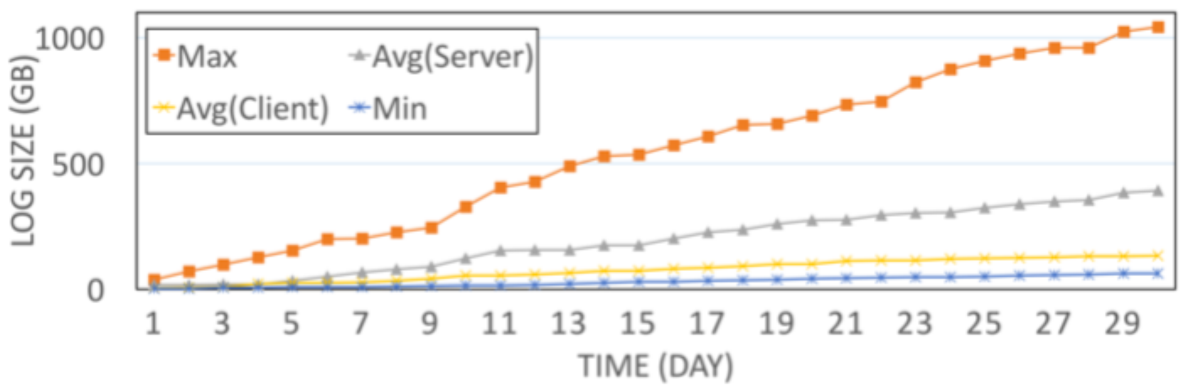
# Evaluation

Multiple real attacks against honeypot ReVirt VM, plus one synthetic attack…

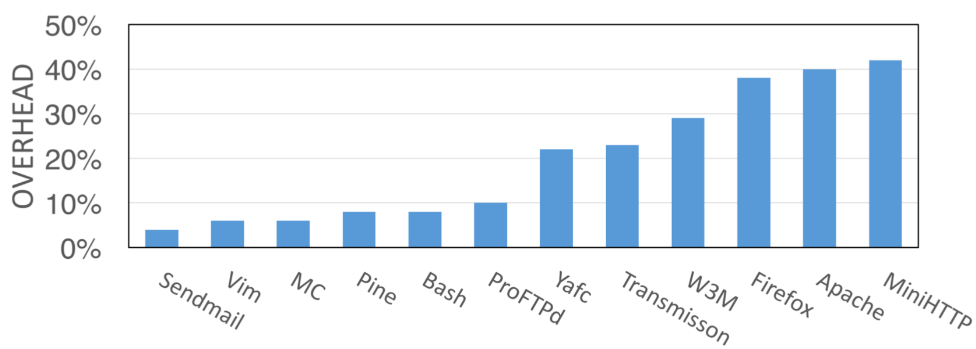
| | *bind* (Fig 5-6) | *ptrace* (Fig 1) | *openssl-too* (Fig 7) | *self* (Fig 8) |
|---|---|---|---|---|
| time period being analyzed | 24 hours | | 61 hours | 24 hours |
| # of objects and events in log | 155,344 objects<br>1,204,166 events | | 77,334 objects<br>382,955 events | 2,187,963 objects<br>55,894,869 events |
| # of objects and events in unfiltered dependency graph | 5,281 objects<br>9,825 events | 552 objects<br>2,635 events | 495 objects<br>2,414 events | 717 objects<br>3,387 events |
| # of objects and events in filtered dependency graph | 24 objects<br>28 events | 20 objects<br>25 events | 28 objects<br>41 events | 56 (36) objects<br>81 (49) events |
| growth rate of EventLogger's log | 0.017 GB/day | | 0.002 GB/day | 1.2 GB/day |
| time overhead of EventLogger | 0% | | 0% | 9% |

# Kernel-Supported Cost-Effective Logging

- BackTracker — still extraordinarily costly
  - In Enterprise environment, one backtrace query may take *days* to return [Liu et al., NDSS'18]
- Ma et al. ATC'18 Linux Audit Benchmarks:



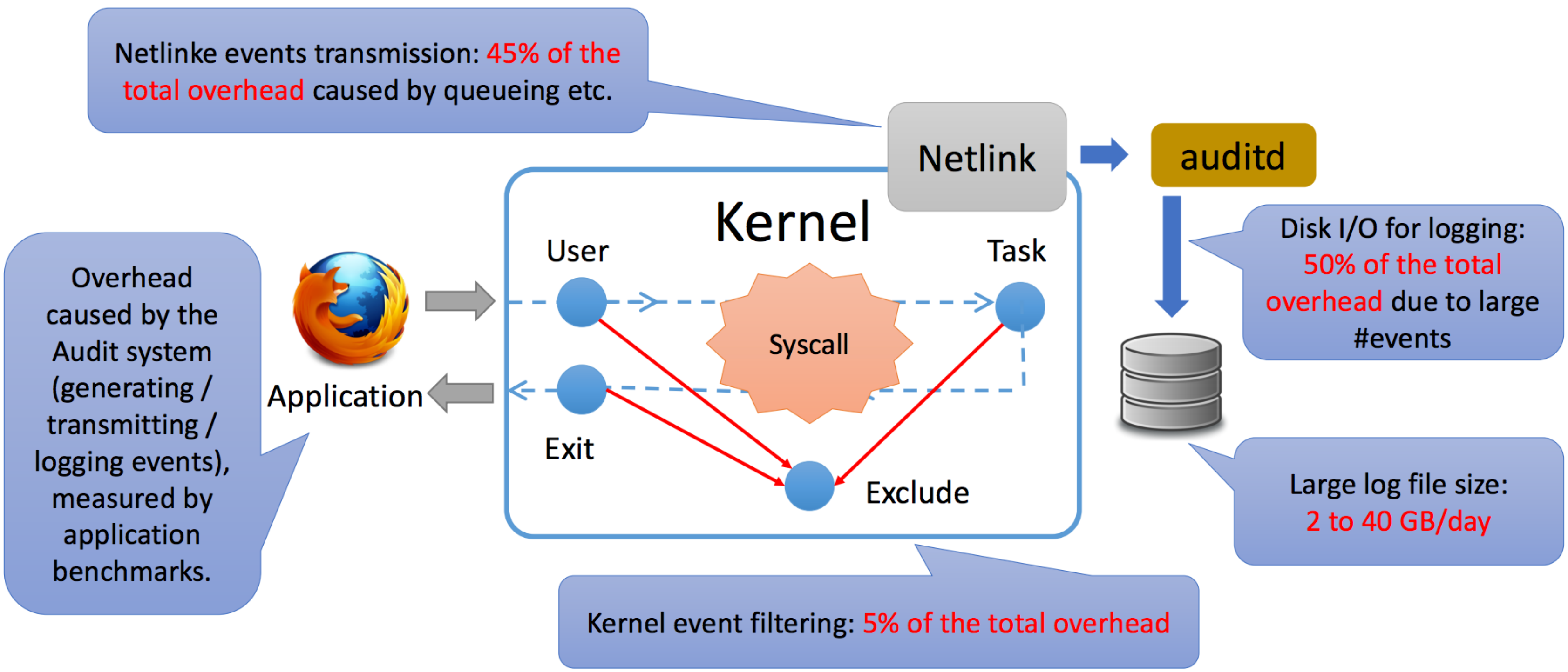

**High Storage Overhead**




**High CPU Overhead**

# Kernel-Supported Cost-Effective Logging

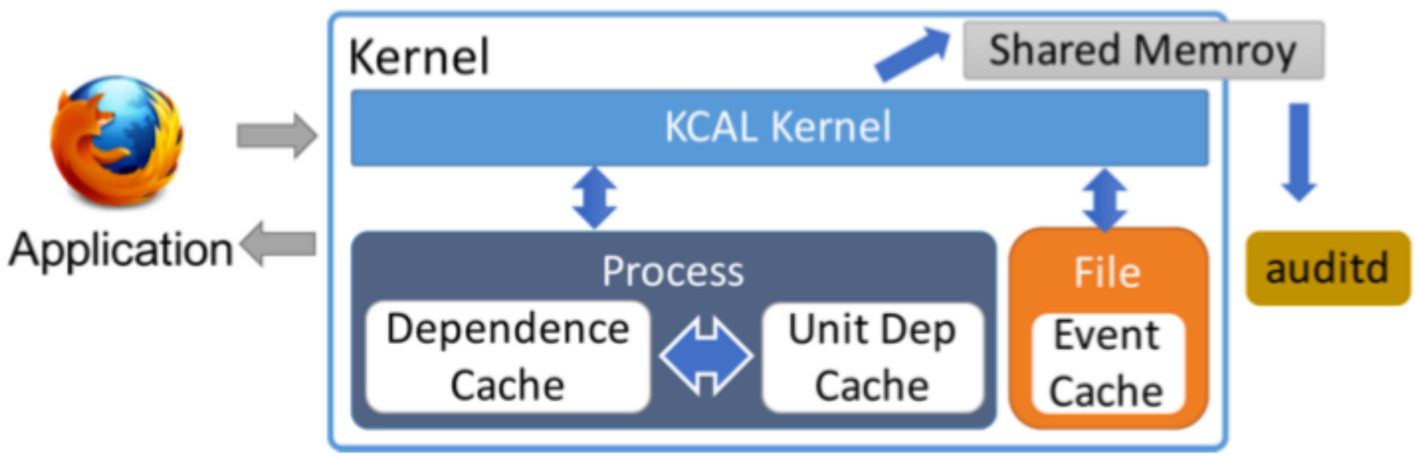
- BackTracker — still extraordinarily costly
  - In Enterprise environment, one backtrace query may take *days* to return [Liu et al., NDSS'18]
- Ma et al. ATC'18 Linux Audit Benchmarks:



# KCAL

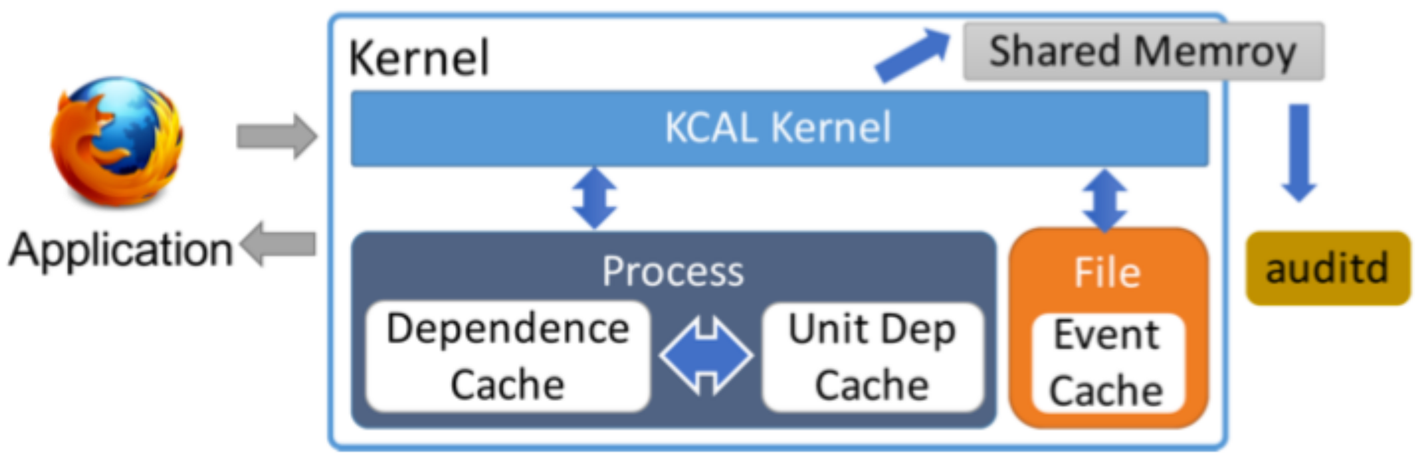
KCAL addresses several shortcomings of Linux Audit

- Raw logging overhead
- In-Kernel execution partitioning
- In-Kernel elimination of event redundancy
- In-Kernel garbage collection of irrelevant events



# KCAL Kernel-User IPC

- KCAL drops inefficient Netlink channel in favor of faster kernel-user communication.
- Uses shared memory instead.
- Same trick used in other auditing frameworks like Hi-Fi (ACSAC'12), LPM (Security'15).



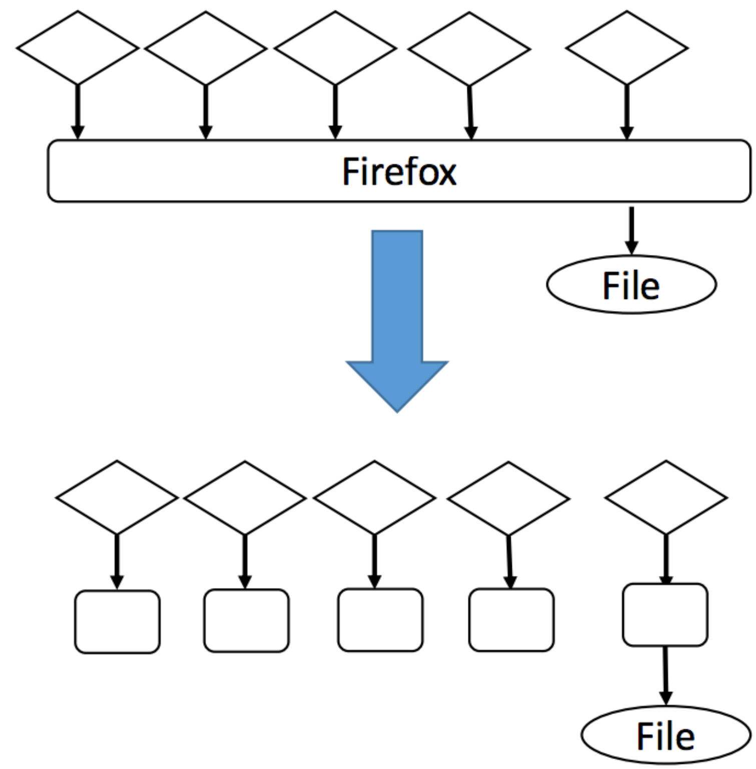
# KCAL Redundancy Filters

- King and Chen 2003 observe event redundancy in offline graph construction phase, eliminate it.
- KCAL pushes redundancy elimination into capture phase
- Achieved through decentralized kernel object cache
- ***Why is it safe to eliminate redundant log events?***

```
1. PID=422, Event = Read (FD4)
2. PID=422, Event = Read (FD4)
3. PID=422, Event = Read (FD4)
4. PID=422, Event = Read (FD4)
5. PID=442, Event = Write(FD5)
6. PID=442, Event = Read (FD4)
7. PID=442, Event = Write(FD5)
8. PID=442, Event = Write(FD5)
9. PID=442, Event = Write(FD5)
```

# KCAL Execution Partitioning

- King and Chen 2003 allude to dependency explosion problem, solve with time slicing
- Dependency Explosion: Each process output assumed to depend on all prior inputs
- KCAL includes execution partitioning* module to address this, enables further reduction




** c.f. BEEP (NDSS'13)*

# KCAL Execution Partitioning

Does EP reduce effectiveness of redundancy filtering?

- No. optimization tracks when one unit's dependency should be applied to addition units.

```
1. PID=442, Event=UNIT_ENTER
2. PID=422, Event=Read (FD4)
3. PID=422, Event=Read (FD4)
4. PID=422, Event=Read (FD4)
5. PID=422, Event=Read (FD4)
6. PID=422, Event=Read (FD4)
7. PID=422, Event=Read (FD4)
8. PID=422, Event=Read (FD4)
9. PID=422, Event=Read (FD4)
10.PID=442, Event=UNIT_EXIT
```

*In-Unit Redundancy*

```
1. PID=442, Event=UNIT_ENTER
2. PID=422, Event=Read (FD4)
3. PID=422, Event=Read (FD4)
4. PID=422, Event=Write(FD5)
5. PID=442, Event=UNIT_EXIT
6. PID=442, Event=UNIT_ENTER
7. PID=422, Event=Read (FD4)
8. PID=422, Event=Read (FD4)
9. PID=422, Event=Write(FD5)
10.PID=442, Event=UNIT_EXIT
```

*Cross-Unit Redundancy*

# KCAL Garbage Collection

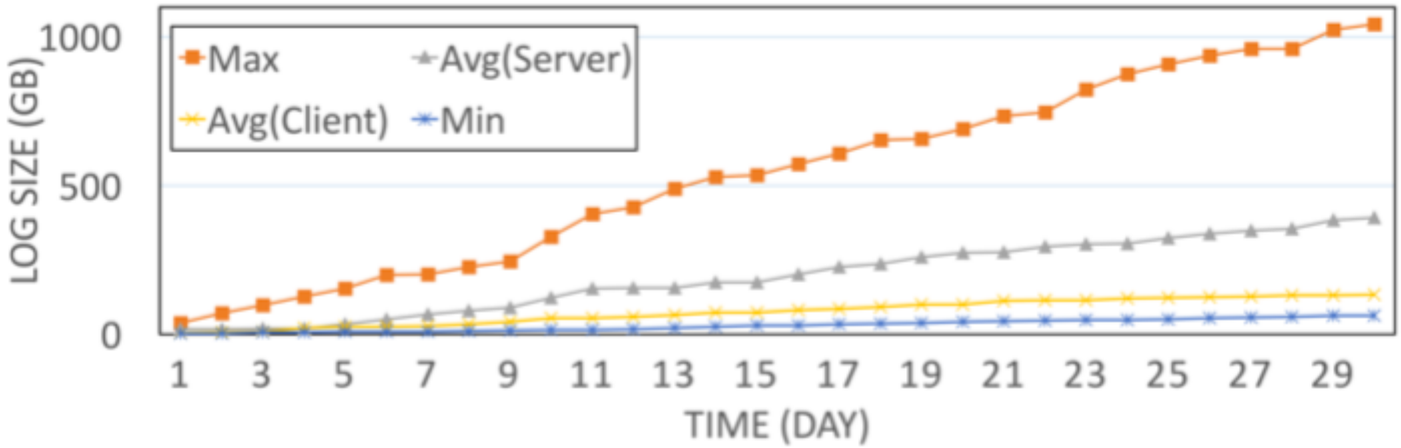
- King and Chen 2003 observe forensically irrelevant files (e.g., read-only) can be filtered.
- KCAL pushes garbage collection into capture phase
- Achieved through decentralized kernel object cache
- ***Why is it safe to eliminate redundant log events?***

```
1. PID=442, Event=UNIT_ENTER
2. PID=422, Event=NewFD(FD5)
3. PID=422, Event=Write(FD5)
4. PID=442, Event=UNIT_EXIT
5. PID=442, Event=UNIT_ENTER
6. PID=422, Event=Write(FD5)
7. PID=442, Event=UNIT_EXIT
8. PID=442, Event=UNIT_ENTER
9. PID=422, Event=Delete(FD5)
10.PID=442, Event=UNIT_EXIT
```

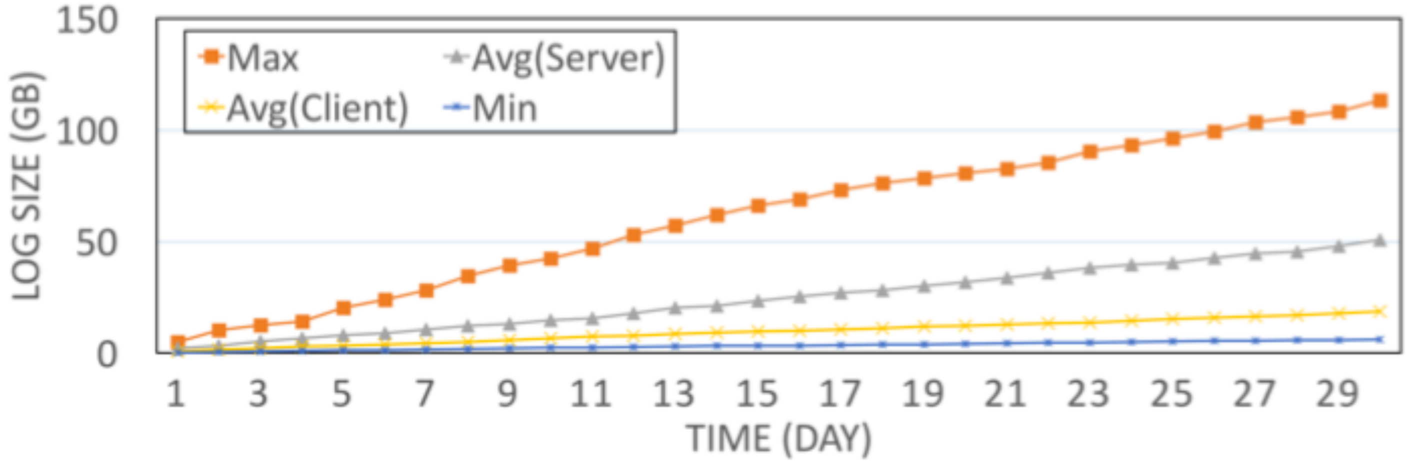
*Temporary files are not relevant to attack forensics*

# KCAL Evaluation

## Storage Overhead



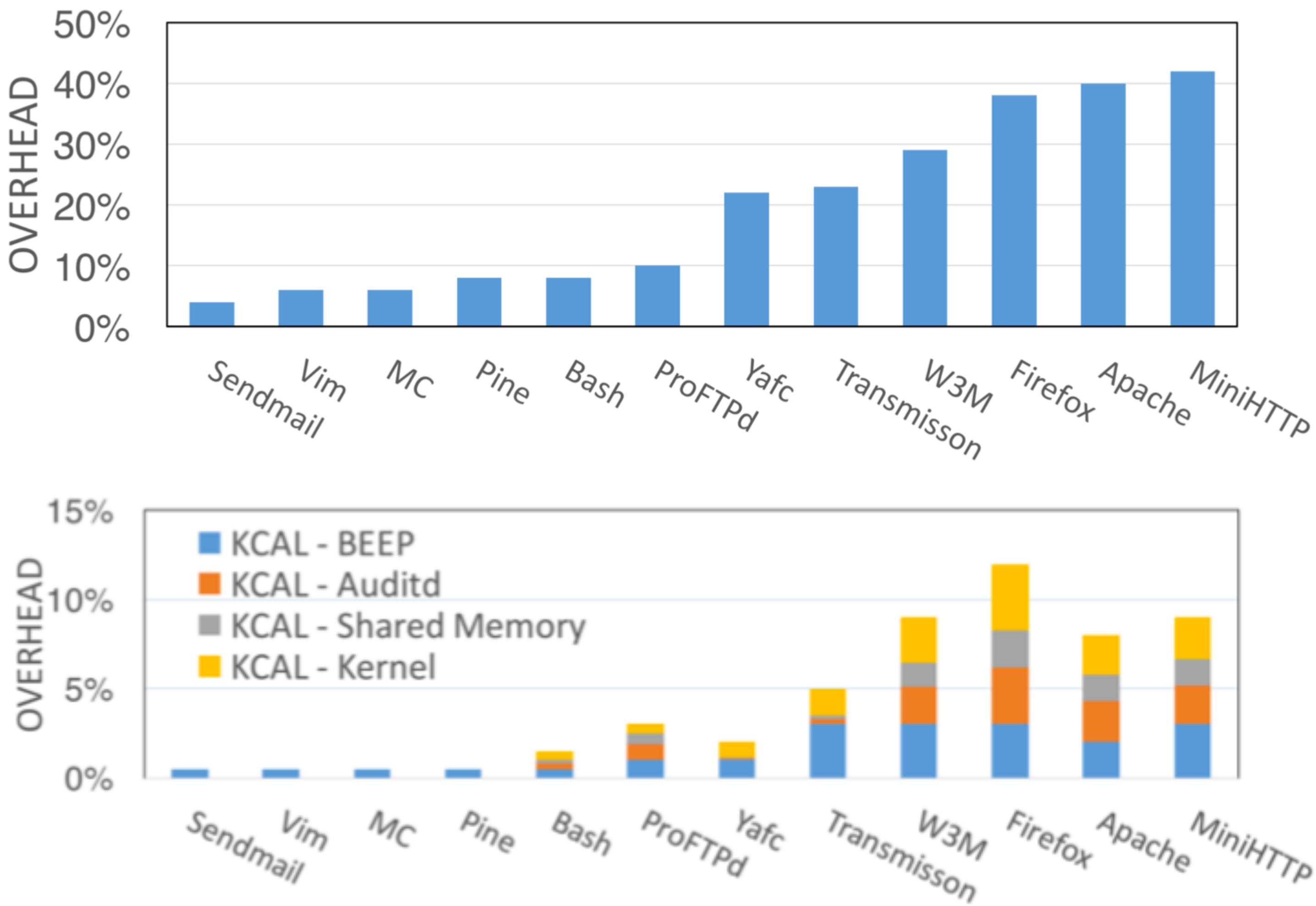

**Before**




**After**

# KCAL Evaluation

## CPU Overhead

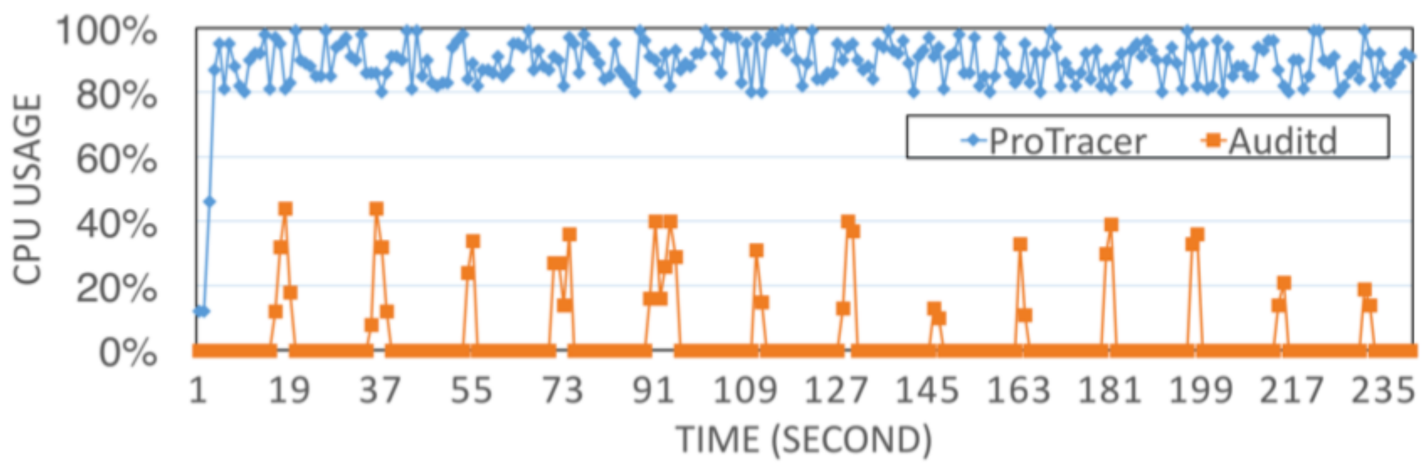



**Before**

**After**

# KCAL Evaluation

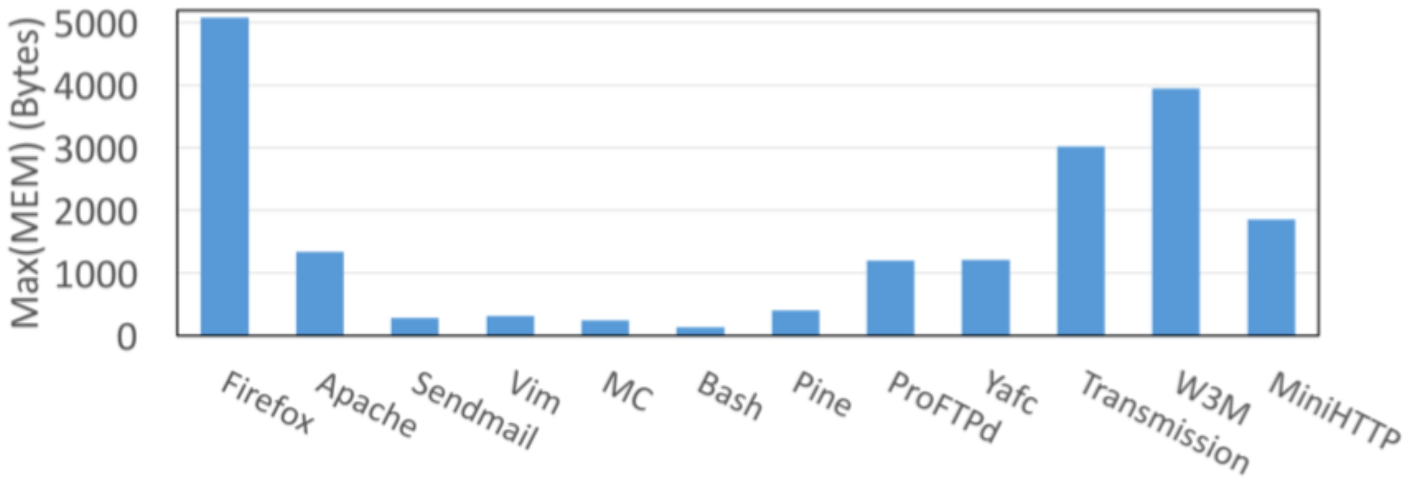
## auditd cpu consumption



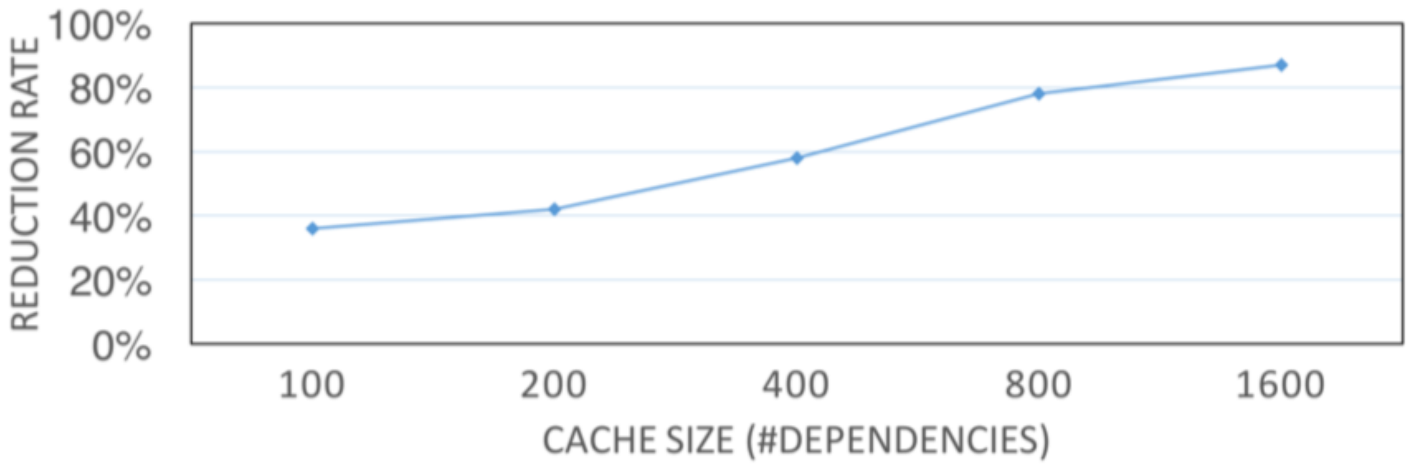

Because kernel is not always logging, `auditd` actually sleeps; normally `auditd` can easily consume 100% of a core's cycles.

# KCAL Evaluation

## Kernel Memory Consumption




**Manageable per-process cache size**




**Graceful degradation as cache size decreases**

# System Intrusions: Looking Forward

- Where to look for literature: “Big 4” security conferences (IEEE S&P a.k.a. Oakland, USENIX Security, CCS, NDSS), reputable second tier conferences (i.e., RAID).
- Hot Topics in System Intrusion (not exhaustive):
  - Attack PROV: Efficiency (e.g., Hybrid Tainting), Fidelity (e.g., Execution Partitioning), Security (e.g., Provenance Monitor)
  - Software Security: Attacks (e.g., any Binary Exploitation stuff), Defenses (e.g., CFI, Privilege Separation, TCB Minimization)
  - Intrusion Detection
  - Vulnerability Discovery (e.g., Fuzzing, Concolic Testing)
  - Network-Based Monitoring and Defense