Determinating Timing Channels in Compute Clouds

Amittai Aviram, Sen Hu, Bryan Ford
Yale University
http://dedis.cs.yale.edu/

Ramakrishna Gummadi
University of Massachussetts Amherst

CCSW, October 8, 2010
Timing Attacks

• **Cooperative attacks** – apply to:
  - Mandatory Access Control (MAC) systems [Kemmerer 83, Wray 91]
  - Decentralized Information Flow Control (DIFC) [Efstathopoulos 05, Zeldovich 06]

• **Non-cooperative attacks** – apply to:
  - Processes/VMs sharing a CPU core [Percival 05, Wang 06, Acıičmez 07, …]
  - Including VM configurations typical of clouds [Ristenpart 09]
Cooperative Attacks: Example

Trojan leaks **secret** information by modulating a **timing channel** observable by **unclassified** app.

**Secret** Level Trojan App

**Unclassified** Level Conspiring App

- use a lot, use a little
- how fast am I running?

MAC/DIFC Protection Boundary

Timeshared Host
Non-Cooperative Attacks: Example

Apps *unintentionally* modulate shared resources to reveal secrets when running standard code!

- **Acme Data, Inc.**
  - Crypto (AES, RSA, ...)

- **Eviltron**
  - Passive Attacker

- **Discretionary Protection Boundary**

- **Cloud Host**

  - key-dependent usage patterns
  - watch memory access timing
The Big Question

Are timing attacks practical in the cloud?

• Answer 1: Maybe. [Ristenpart 09]

• Answer 2: I don't know.

Answer is not the subject of this talk.
The Other Big Question

“Attacks never get worse; they only get better.”
- NSA?

If timing attacks become practical in the cloud, what can we do about them?
Talk Outline

- The Timing Channel Problem
- **Why They're Worse in the Cloud**
- A Deterministic, Timing-Hardened Cloud
- Feasible? A Bit of Evidence
  - (preliminary performance results)
- Conclusion
Why Pick On Cloud Computing?

Cloud computing **exacerbates vulnerabilities:**

1. Mutually distrustful tasks *routinely co-resident*
2. Clouds introduce *massive parallelism*
3. Cloud-based timing attacks *won't get caught*
4. Partitioning defeats *elasticity of the cloud*
1. Routine Co-Residency

On Private Infrastructure:
- Owner can manage all running software
- Attacker must get code installed locally (e.g., malware) before starting attack

On Cloud Infrastructure:
- Provider doesn't manage running guest apps
- Attacker simply buys CPU time to run attack
- No protection compromised → no alarms
2. Massive Parallelism

- All shared resources create timing channels
  - CPUs, caches, interconnects, I/O devices, …
- Cloud jobs use *many* resources in parallel
  - Multiply attack surface by $N$
3. Timing Attacks Won't Get Caught

On Private Infrastructure:
- Owner can *monitor* all running software (antiviral software, intrusion analysis, …)

On Cloud Infrastructure:
- Customer A *cannot* monitor customer B's apps
- Provider *can*, but wouldn't want to
  - Not their job to ask questions
  - Might invite privacy lawsuits
4. Partitioning is Infeasible

Current timing hardening approaches are either:

- **Specific to particular algorithms & resources**
  - Equalize AES path lengths, cache footprint, …
- **General but contrary to cloud business model**
  - **Partition** CPU cores, cache, interconnect, …
  - Can't oversubscribe, stat-mux resources
  - **Cloud loses its elasticity!**
Talk Outline

- The Timing Channel Problem
- Why They're Worse in the Cloud
- A Deterministic, Timing-Hardened Cloud
- Feasible? A Bit of Evidence
  - (preliminary performance results)
- Conclusion
Anatomy of a Timing Channel

Two elements required: [Wray 91]

- A resource that can be modulated by the signaling process (or victim)
- A reference clock enabling the attacker to observe, extract the modulated signal

Remove either → no timing channel.
Prior Approaches

Attempt to **eliminate modulation**
- e.g., by partitioning hardware resources
Our Approach

Allow modulation, **eliminate reference clocks**

Customer A's Job

Customer B's Job
Our Approach

Allow modulation, **eliminate reference clocks**

- *Dynamic statistical multiplexing allowed*
Deterministic Execution

Definition:
- Given *same inputs* from external world
- Always yields *same execution flow & outputs*

What this means:
- Execution not affected by *internal* timing
- No *internal* reference clocks (only external)
A Timing-Hardened Cloud

Deterministic Cloud

Customer A's Job

- Storage Node
- Compute Node

Customer B's Job

- Storage Node
- Compute Node

Gateway

Job Inputs
Job Outputs

Job Inputs
Job Outputs
What We've Accomplished

Eliminated all *internal* timing channels

- Independent of **resource** (cache, disk, …)
- Independent of **algorithm** (AES, RSA, …)

Leaves **one** aggregated timing channel

- How long did the entire job take to run?

Can **rate control** by scheduling job outputs
Eliminating Reference Clocks

Just protect hardware clocks/timers from apps.

*Easy, right?*

*Wrong.*
A Thread is a Reference Clock

volatile long long timer = 0;

void *timer_func(void *)
    { while (1) timer++; }

main() {
    pthread_create(&timer_thread, NULL,
                  timer_func, NULL);
    ...
    // Read the "current time"
    long long long timestamp = timer;
    ...
}
Deterministic Parallelism

Requires **new approach to parallel execution**

- Threads *access memory* deterministically
- Threads *synchronize* deterministically
- Processes *access shared system resources* (e.g., file systems) deterministically

→ Parallelism introduces **no reference clocks**,
→ Hence **no internal timing channels**
Introducing Determinator

A Determinism-Enforcing Microkernel/Hypervisor
- “Efficient System-Enforced Parallelism”
  (Jay Lepreau Best Paper Award, OSDI 2010)
- Explores a new, naturally deterministic parallel application programming model

Other Approaches
- DMP/CoreDet/dOS [Bergan 2009/2010]
- Grace [Berger 2009]
Determinator Architecture

A Determinism-Enforcing Microkernel/Hypervisor

Diagram showing the architecture with layers labeled Root Space, Child Space, Grandchild Space, and Hardware.
Other Benefits of Determinism

Simpler Application Development/Debugging
- No races/heisenbugs $\rightarrow$ all bugs repeatable

More efficient logging/replay
- Log only external, not internal events

State machine replication, checking, analysis
- Bit-for-bit correspondence across replicas
Are Deterministic Clouds Practical?

Determinism *could* help control timing channels, *but*:

- Can it offer a **rich enough environment**?
- Can it be made **efficient enough**?

Some open issues and possible solutions...
Can It Be Efficient Enough?

Some preliminary evidence...

- (see OSDI paper for more detailed evaluation)
Creating a Rich Cloud Environment

Sometimes apps need to tell the time

- External nodes or gateways supply timestamps as explicit, external inputs

May be some forms of “safe nondeterminism”

- Random numbers from provider's trusted RNG

Sometimes want application-level scheduling

- App can fork off “scheduler process,” but use IFC to prevent it from affecting app's results
Conclusion

• Timing channels *may* be a serious challenge
  – Clouds create *massive untrusted co-residency*
  – Parallelism creates *pervasive timing channels*
  – Timing attacks are *unlikely to be caught*
  – Resource partitioning *defeats business model*

• Deterministic parallelism may offer a solution
  – Eliminates all *internal* timing channels
  – Performance practical at least for some apps

Further information: http://dedis.cs.yale.edu