Deterministically Deterring Timing Attacks in Deterland

Weiyi Wu, Ennan Zhai, Daniel Jackowitz, David Isaac Wolinsky, Liang Gu, Bryan Ford

Yale University


University of Massachusetts Amherst
April 28, 2015
Talk Outline

• Background: Attacks and Mitigation in the Cloud
• Design: Hypervisor-Secure Mitigation
• Implementation: Deterland Hypervisor
• Evaluation: It Works (at a Cost)
• Conclusion
Talk Outline

• **Background: Attacks and Mitigation in the Cloud**
  - The Timing Attack Problem
  - Why the Problem is Worse in the Cloud
  - Known Approaches to Mitigation

• **Design: Hypervisor-Secure Mitigation**

• **Implementation: Deterland Hypervisor**

• **Evaluation: It Works (at a Cost)**

• **Conclusion**
Timing Attacks via Shared Hardware Resources

cloudcomputingnode

victimVM

cache

evilVM

secret
Classes of Timing Attacks

- **Internal or Local Attacks:**
  - Attacker has full control over guest VM co-resident with victim
  - Attack VM carries out attack autonomously from within the cloud environment

- **External or Remote Attacks:**
  - Attacker has limited/no control over guest VM
  - Attacker must carry out attack (partly) remotely from outside the cloud environment
Internal Attacks: Simplified Example

- **Attacker VM**
  1. Run code in target cacheline
  2. Wait
  3. Run again, *measure time*

- **Victim VM**
  If (S) {
  code touching target cacheline
  }

**Report:**
- S=0 if quick
- S=1 if slow

**Shared Resource, e.g., cache**
External Attacks: Simplified Example

Attack
Client send request

Victim VM

Shared Resource, e.g., Network

Service request:
If (S) {
  compute something that takes time
}

Measure Delay
S=0 if quick
S=1 if slow

Victim Server
S = Secret
(1 bit)
Demonstrated Attacks

- Internal/Local attacks naturally easier
  - Through *many* resources:
    - L1 code cache, L1 data cache, function units, branch target cache, last-level cache, ...
  - Including cross-VM attacks in cloud environments
    [Zhang'12, Yarom'13, Irazoqui'14, ...]

- But External/Remote attacks demonstrated too
  - e.g, remotely steal private RSA keys from non-constant-time SSL/TLS libraries
    [Bonneau'06, Brumley'10, Chen'10, ...]
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 $\rightarrow$ 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 $N$

![Yesterday](image1)

![Today](image2)
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 probably doesn'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!**
Timing Channel Mitigation

Timing channels require: [Wray 91]

- A resource that the victim process may (inadvertently) modulate
- A reference clock enabling the attacker to observe, extract the modulated signal

Remove either → no timing channel.
Approach 1: Eliminate Modulation

(a) by statically partitioning hardware resources
- Generalizes over code, must modify hardware
- Incompatible with cloud business model

Split Resource, e.g., cache
Approach 1: Eliminate Modulation

(b) via constant-time code execution

- General **hardware**, but specialized **code**
Approach 2: Deny Reference Clocks

If attack VM can't **tell time**, can't **measure time**

- At least not locally, **internal** to cloud

![Diagram showing Attacker VM with crossed-out clock and Victim VM with shared resource, e.g., cache.]

**Shared** Resource, e.g., cache
Approach 2: Deny Reference Clocks

Attacker can still measure time remotely

- But we **mitigate** to rate-limit external leakage
Deterministic Mitigation

- Variants proposed independently by:
  - [Aviram'10] – Determinator basis, cloud focus
  - [Askarov'10] – PL basis, formal analysis
  - [Stefan'12] – PL basis, Haskell/Monads prototype

- No prior prototype of general mitigation compatible with existing apps & Oses
  - That's what's new!
Talk Outline

• Background: Attacks and Mitigation in the Cloud

• **Design: Hypervisor-Secure Mitigation**
  – Deterland cloud and hypervisor architecture
  – Hypervisor-enforced mitigation, step-by-step

• Implementation: Deterland Hypervisor

• Evaluation: It Works (at a Cost)

• Conclusion
Deterland Cloud Architecture

- Cloud provider offers different classes of VMs with different timing mitigation parameters
  - Only VMs with **same mitigation parameters** directly share physical machines
Deterland Hypervisor

- Based on CertiKOS
  - In turn based on Determinator
- Designed to be simple, formally verifiable hypervisor
  - CertiKOS is largely verified, but Deterland isn't (yet)
Deterland Hypervisor Architecture

- VM
- hypervisor
  - vTimer
  - simulated devices
  - virtio devices
- monitor
- mitigator
- backend
  - drivers
  - physical I/O devices

mitigation boundary

artificial time

wall time
Mitigation Operation: Overview

- Start with **unrealistic** simplifying assumptions
  - Need “batch” processing jobs only (mainframe-era)
  - Can predict job's worst-case execution time
- Then relax, make more general/realistic
  - Quantize time into coarse-grained **ticks**
    - Secure deterministic execution *between ticks*
    - Inputs accepted, outputs produced only at ticks
    - Virtual CPU speed trades efficiency vs leakage risk
  - For compatibility: use **virtio** for all mitigated I/O
  - For throughput: batch virtio inputs/outputs each tick
Overly-Simplified Example

- Batch operation, known worst-case exec time
  - Attacker submits input $I$, cloud computes pure $f(I)$, always returns result *exactly* 1 “clock-tick” later because $f$ limited to (say) 1M instructions

Input $I$ at tick $t$

Mitigated I/O: Network, Disk, …

Output $O$ at tick $t+1$

Attackers VM

Victim VM

if(S)

Shared Resource, e.g., cache
Overly-Simplified Example

Intuitive reasoning (formalized by Askarov):

- Attacker can learn leaked info only via either content of output $O$ or timing of its production
  - If $O$ is a pure function of its explicit input, $O = f(I)$, then $O$ cannot depend on nondeterministic timing
    - Principle: determinism closes internal timing channels
  - If $O$ is always produced after the same delay, then timing of $O$ cannot reveal any information
    - Principle: constant delay closes external channels
What Type of Determinism?

- **Weak Determinism:** typically library-implemented, works on *race-free* code
  [Grace, Kendo, ...]

- **Strong Determinism:** typically library-implemented, works on *non-malicious* code
  [CoreDet, Dthreads, ...]

- **Secure Determinism:** system-enforced, works on *adversarial* code
  [Determinator, Deterland]
Deterland requires Secure, System-Enforced Determinism

- If attacker-controlled VM can escape determinism enforcement, attacker can tell time → high-rate internal timing channel leak
- Most any source of nondeterminism is usable, e.g., launch thread that increments-and-spins
- Deterland must
  - Prevent unsynchronized cross-thread interaction
  - Prevent malicious escape from deterministic sandbox

```java
int bogoTime = 0
thread QuasiTimer {
    while (true) {
        bogotime++
    }
}
```
From Batch to Interactive

**Intuition:** “interactive operation” is just a series of small batch operations

- Cloud customer (e.g., attacker) can submit **one** new “batch input” per regular “clock tick”
  - Safe to maintain guest VM state across ticks
  - Safe to combine several inputs into one clock tick

\[
\begin{align*}
\text{Inputs } I_1 & \text{ at } t \\
\text{Outputs } O_1 & \text{ at } t+1 \\
\text{Inputs } I_2 & \text{ at } t+1 \\
\text{Outputs } O_2 & \text{ at } t+2 \\
S_0 & = f(I_1, S_0) \\
S_1 & = f(I_2, S_1) \\
S_2 & \\
\end{align*}
\]
Relax Worst-Case Execution Time

- Don't require every input to be done in 1 tick
  - “Easy-to-execute” ticks waste CPU capacity
- Instead, output delay is integral number of ticks
  - Extra ticks are “bubbles”, which can leak info
  - But can leak at most one bit per tick
Mitigation in Deterland Hypervisor

- Runs unmodified legacy OSes and applications
- All I/O uses standard **virtio** interface
  - Each tick, guest gets a new batch of virtio inputs, and produces a new batch of virtio outputs
  - General: mitigates all devices with virtio support
- Configurable **clock tick period**
  - Short better for I/O, long better for CPU efficiency
- Configurable **instructions-per-tick**
  - Fewer better for leakage, more better for efficiency
Talk Outline

- Background: Attacks and Mitigation in the Cloud
- Design: Hypervisor-Secure Mitigation
- Implementation: Deterland Hypervisor
- Evaluation: It Works (at a Cost)
- Conclusion
Implementation Summary

- Works, runs unmodified Linux (Ubuntu) guests
  - Deterministically emulates PIT, RDTSC timing
  - Virtio-based disk, network devices supported
- Limitation (inherited from CertiKOS): currently only one guest VM per physical core
  - Not fundamental, just per-core scheduler missing
- Limitation: one virtual core per guest VM
  - Much harder to solve efficiently, deterministically
Counting Instructions

• Challenge: x86 hardware can't trigger precise exception or VMExit after given # of instructions
  – Solution: imprecise performance counters plus single-stepping from “undershoot” to exact point
  – Classic technique used in ReVirt, etc.

• Works, but slow: major CPU cost per trigger
  – Amortizable if Deterland clock ticks are long, but long clock ticks are bad for I/O latencies
  – Historical architectures (e.g., PA-RISC) had precise instruction-counting; maybe future CPUs could too?
Talk Outline

- Background: Attacks and Mitigation in the Cloud
- Design: Hypervisor-Secure Mitigation
- Implementation: Deterland Hypervisor
- Evaluation: It Works (at a Cost)
- Conclusion
PARSEC Benchmark Results

1ms 100ms

speedup ratio

0 0.2 0.4 0.6 0.8 1

blackscholes canneal fluidanimate streamcluster swaptions
Database Benchmark Results

![Graph showing benchmark results for different databases and operations.

- Remote, ext4
- Remote, btrfs
- Local, ext4
- Local, btrfs

- Cassandra 50/50 read/update
- Cassandra read-only
- Cassandra read-modify-write
- MySQL TPC-C

Speedup ratio vs. time (ms).]
Potential Future Optimizations

- Currently *all* I/O is mitigated, may be overkill:
  - *Intra-cloud, inter-guest communication:* could use deterministic messaging, avoid mitigation
  - *Intra-cloud disk access:* could offer deterministic read/write, avoid mitigation

- Mitigate at higher level of abstraction:
  - Mitigate sockets, move TCP stack out of guest VM → avoid bad mitigation effect on congestion control
  - Mitigate files, move filesystem out of guest VM → avoid bad mitigation effect on sync writes, etc.
Compiler/Hardware Opportunities

• Deterministic instruction counting is costly
  – Potential alternative: lightweight code rewriting?
  – Long-term: why oh why doesn't hardware do this?

• Instruction count is also a poor model for “deterministic time”
  – Falsely pretends all instructions about equally hard
  – Potential alternative: deterministic cost models?
  – Long-term: hardware support for cost models?
Conclusion

- First proof-of-concept of timing channel mitigation for existing unmodified OSes, apps
- General I/O mitigation model for virtio devices
- Usable performance for CPU-intensive loads, currently high costs for I/O-intensive loads
- Just first step, many improvements possible

Yale DeDiS group: [http://dedis.cs.yale.edu](http://dedis.cs.yale.edu)