Deterministically Deterring Timing Attacks in Deterland

Weiyi Wu,
Ennan Zhai, Daniel Jackowitz,
David Isaac Wolinsky, Liang Gu
Yale University

Bryan Ford
EPFL

TRIOS – October 4, 2015
Timing Attacks via Shared Hardware Resources
Talk Outline

• Background: Attacks and Mitigation in the Cloud
• Design: Hypervisor-Secure Mitigation
• Implementation: Deterland Hypervisor
• Preliminary Results: It Works (at a Cost)
• Conclusion
Timing Attack Background

• **Internal or Local Attacks:**
  - Attacker controls VM co-resident with victim
  - Operates from *within* the cloud environment

• **External or Remote Attacks:**
  - Attacker has *limited/no* control over guest VM
  - Operates from *outside* the cloud environment
  - Brumley/Boneh, “Remote timing attacks” 2005
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
}

Shared Resource, e.g., cache

Attacker Code

Victim Code
S = Secret (1 bit)

Report:
S=0 if quick
S=1 if slow
External Attacks: Simplified Example

Victim VM

S = Secret (1 bit)

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

Victim Server

S = Secret (1 bit)

Shared Resource, e.g., Network

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

Client

Attack

send request
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*

Aviram et al., “Determinating Timing Channels in Compute Clouds” [CCSW '10]
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

Attacker VM

Victim VM

if(S)
Approach 1: Eliminate Modulation

(b) via constant-time code execution

- General **hardware**, but specialized **code**
- Difficult to write, broken by “smart” compilers
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 Deny Reference Clocks approach]

- **Attacker VM**
- **Victim VM**
- **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
Talk Outline

• Background: Attacks and Mitigation in the Cloud

• Design: Hypervisor-Secure Mitigation
  – Timing-Channel Mitigation Overview
  – System-enforced Determinism in Deterland
  – Practical hypervisor-enforced mitigation

• Implementation: Deterland Hypervisor

• Preliminary Results: It Works (at a Cost)

• Conclusion
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 $t$

Output $O$ at tick $t+1$

Mitigated I/O: Network, Disk, …

Attacker 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]
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]
Mitigation 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++
    }
}
```
Deterland Hypervisor

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

![Diagram of cloud architecture with different classes of VMs:]
- Data protection
- Deterministic execution
- Low leakage rate
- Medium leakage rate
- Non-deterministic execution
- Unconstrained leakage rate
Mitigation for Interactive I/O

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

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

• Background: Attacks and Mitigation in the Cloud
• Design: Hypervisor-Secure Mitigation
• Implementation: Deterland Hypervisor
• Preliminary Results: 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

• Workaround: “scale-out” across many single-core guests on each multi-core machine
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
● Preliminary Results: It Works (at a Cost)
● Conclusion
CPU-intensive Microbenchmark

Otherwise we pay the “real-time tax” of underutilization.

Load overhead if we keep the CPU busy.
Performance vs Leakage Bound

The diagram illustrates the relationship between vCPU frequency (GHz) and slowdown ratio or leakage rate. The graphs compare precise and coarse models, with specific timescales (1ms, 100ms) indicated for each model type. The slowdown ratio and leakage rate are measured across different vCPU frequencies (0.8 to 6.4 GHz). The precise models show lower slowdown ratios and leakage rates compared to the coarse models, particularly at higher vCPU frequencies.
Real Compute-intensive Workloads

Upshot: not too bad, if we keep the CPU busy
- Mitigation hurts I/O-intensive work (of course)
  - Heavily dependent on mitigation interval
  - Possible solution: deterministic disk/FS access
Main problem: mitigation of guest TCP stack
- Congestion control highly sensitive to timing
- Possible solution: move TCP stack out to hypervisor
Potential Future Optimizations

- Mitigating all I/O is unnecessary in principle:
  - Deterministic intra-cloud, inter-guest networking
  - Deterministic intra-cloud disk access

- Mitigate at higher levels of abstraction:
  - Move TCP, congestion control out of guest VM
  - Move filesystem, disk drivers out of guest VM

- Determinate but don't mitigate:
  - Enforced determinism alone eliminates *local* attacks
  - Mitigation needed only to rate-limit *remote* attacks
    - Can disable if remote attack risk is deemed remote
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 hypervisor implementing 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

More info: http://dedis.cs.yale.edu/cloud/

Code: git@dedis.cs.yale.edu:verikos tifc rtl