System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud

Taesoo Kim, Marcus Peinado, Gloria Mainar-Ruiz

MIT CSAIL       Microsoft Research
Security is a big concern in cloud adoption

Security still the ‘No. 1 obstacle’ to cloud adoption

By Bobbie Johnson | Jun. 20, 2012, 10:17am PT | 2 Comments

International companies are still wary of cloud adoption because of concerns over data security and legal exposure, including worries about American government interference.

Speaking at GigaOM’s Structure 2012 conference in San Francisco, Juergen Urbanski of Deutsche Telekom’s T-Systems said that European customers, in particular, were wary of moving to the cloud because of their security fears.

(L to R) Juergen Urbanski of T-Systems, Tony Lucas of Flexiant, Steve Collen of Huawei
(c) 2012 Pinar Ozger pinar@pinarozger.com
Why are cache-based side channel attacks important?

- CPU cache is the most fine-grained shared resource in the cloud environment

- Cache-based side channel attacks:
  - 2003 DES by Tsunoo et al. (with $2^{26.0}$ samples)
  - 2005 AES by Bernstein et al. (with $2^{18.9}$ samples)
  - 2005 RSA by Percival et al. (-)
  - ...
  - 2011 AES by Gullasch et al. (with $2^{6.6}$ samples)
Background: CPU & Memory
Background: cache structure

Core1
Core2
Core3
Core4

L3

Cache hit
~50

Cache miss
~240

RAM

8M

16G

x 2046
Background: cache terminologies

- **Pre-image set**: set of memory mapped into the same cache line
Background: cache terminologies

- **Pre-image set**: set of memory mapped into the same cache line
- **Cache line set**: set of cache lines mapped by the same pre-image set

![Diagram of cache line set and pre-image set with colored pages](image)

Different class of colored pages
Background: cache-based side channel

Victim
Core1
8M

Attacker
Core2

Core3

Core4

~50

~240

Cache hit

Cache miss

L3

RAM

16G
Cache-based side channel attacks (cache attacks)

```c
while(1) {
    beg = rdtsc();
    access memory
    diff = rdtsc() - beg
}
```
Types of cache attacks

• **Time-driven attacks**
  : measure *access time* depending on states of cache

• **Passive** time-driven attacks
  : measure total execution time of victim

• **Active** time-driven attacks
  : manipulate states of cache

• **Trace-driven attacks**
  : probe which cache lines victim has *accessed*

  → Attackers should **co-locate** with a victim
Goal

To provide cloud tenants a protection mechanism against cache attacks:

- Active time-driven attacks
- Trace-driven attacks

But our solution still provides:

- Minimal performance overhead
- Compatible with commodity hardware
Idea: protect only sensitive data

- Give a private page to each cloud tenant
  - No other tenants can cause cache interference

- Load sensitive data to the private page

```c
void *sm_alloc(size_t size)
void sm_free(void *ptr)
```
Strawman: construct a private page

- **Do not assign** pre-image sets of the private pages (same colored pages) to other VMs
Strawman: assign a private page to each VM

1. How to make sure that a private page stays in the cache?
Strawman: assign a private page to each VM

2. How to make it scalable if we increase the number of VMs?
Strawman: assign a private page to each VM

3. How to utilize the reserved regions?

~1% x 5
Three challenges

1. How to make sure that a private page stays in the cache?
   → **Lock cache lines**

2. How to make it scalable if we increase the number of VMs?
   → **Assign** a private page **per core**

3. How to utilize the reserved regions?
   → **Mediate accesses** on reserved regions
1. Locking cache lines

• **Locked**: never evicted from the cache

• **Inertia** property of cache (shared LLC):
  • An **eviction** only can happen when there is an **attempt to add** another item into the cache
  • Cache lines will **stay** still until we **access** an address that is **not in the cache**
Cache interference

- Context switches
- Hyperthread
- Simultaneous execution
Keep cache lines locked

- Context switch:
  - **Reload** locked cache lines
- Hyperthread:
  - Force **gang** schedule
    - (no two VMs run on the same core simultaneously)
- Simultaneous execution:
  - **Never map** pages that **collide** with private pages
2. Assign a private page per core

- Load a private page of active VM onto the private page of the core
2. Assign a private page per core

- No cache interference between running VMs
Save / load private pages on context switch
3. Utilize reserved regions

- **Assign** pages to VMs
- **Mediate** their accesses
Page Table Alert (PTA)

- Mark **invalid** on reserved pages (pre-image sets)
- Mediate their accesses in the **page fault** handler
Handle Page Table Alert (PTA)

Set-associativity (w=3)

Pre-image set
Summary of design

• Tenants use a private page for sensitive data

• Assign a private page per core
  • Use fixed amount of reserved memory
  • Load a private page of VM on one of the core

• Utilize reserved regions
  • Assign reserved regions to VMs as usual
  • Mediate their accesses with PTA
Implementation: **StealthMem**

- **Host OS:** Windows Server 2008 R2
  - `bcdedit`: configure **reserved** area as **bad pages**

- **Hypervisor:** **HyperV**
  - Disable **large pages** (2MB/4MB)
  - Mediate `invd`, `wbinv` instructions from VMs
  - Expose a **single private page** to VM

<table>
<thead>
<tr>
<th>Component</th>
<th>Modified lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootmgr/Winloader</td>
<td><strong>500</strong> lines of C</td>
</tr>
<tr>
<td>HyperV</td>
<td><strong>5,000</strong> lines of C</td>
</tr>
</tbody>
</table>
Evaluation

• How much overhead?
  • How does it compare with the stock HyperV?
  • How does it compare with other mechanisms?
  • How to understand overhead characteristics?

• How easy to adopt in existing applications?
  • How to secure popular block ciphers?
Overhead without large pages

Run Spec2006

Stealth w/o large pages: -5.9%

Average: -4.9%

Execution time (s)
Compare with PageColoring

- **PageColoring**: statically divide caches per VM
- **Run SPEC2006 with various #VM**

---

**StealthMem**

<table>
<thead>
<tr>
<th>Application</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
</tr>
<tr>
<td>milc</td>
<td></td>
</tr>
<tr>
<td>namd</td>
<td></td>
</tr>
<tr>
<td>dealII</td>
<td></td>
</tr>
<tr>
<td>soplex</td>
<td></td>
</tr>
<tr>
<td>povray</td>
<td></td>
</tr>
<tr>
<td>calculix</td>
<td></td>
</tr>
<tr>
<td>astar</td>
<td></td>
</tr>
<tr>
<td>wrf</td>
<td></td>
</tr>
<tr>
<td>sphinx3</td>
<td></td>
</tr>
<tr>
<td>xalancbmk</td>
<td></td>
</tr>
</tbody>
</table>

---

**PageColoring**

<table>
<thead>
<tr>
<th>Application</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
Microbench: overheads with various working sets

- Microbench:
  - Working set: vary array size between 1~12 MB
  - Read array in quasi-linear fashion
  - Measure execution time

- Settings:
  - Each VM has a private page
  - 7 VMs: one VM runs microbench while others idle
    - Baseline, PageColoring
    - StealthMem (w/o PTA): do not utilize reserved regions
    - StealthMem (w/ PTA): utilize reserved regions with PTA
Microbench: overheads with various working sets

![Graph showing execution time vs. working set size with TLB: 2MB = 4KB x 512 and L3: 8MB]
Microbench: overheads with various working sets
Modifying existing applications

- e.g., modify **Blowfish** to use StealthMem

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Size of S-box</th>
<th>LoC changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>256 * 8 = 2 kB</td>
<td>5 lines</td>
</tr>
<tr>
<td>AES</td>
<td>1024 * 4 = 4 kB</td>
<td>34 lines</td>
</tr>
<tr>
<td>Blowfish</td>
<td>1024 * 4 = 4 kB</td>
<td>3 lines</td>
</tr>
</tbody>
</table>

Original code:
```
static unsigned long S[4][256];
```

Modified code:
```
typedef unsigned long ULA[256];
static ULA *S;

<@initialization function>
S = sm_alloc(4*4*256);
```
Overhead of secured ciphers

- Encryption **throughput** of DES / AES / Blowfish
  - Baseline: unmodified version
  - Stealth: **secured S-Box** with StealthMem

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Baseline</th>
<th>Stealth</th>
<th>A small buffer (50,000 bytes)</th>
<th>Baseline</th>
<th>Stealth</th>
<th>A large buffer (5,000,000 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>60 MB/s</td>
<td>58</td>
<td>-3%</td>
<td>59 MB/s</td>
<td>57</td>
<td>-3%</td>
</tr>
<tr>
<td>AES</td>
<td>150 MB/s</td>
<td>143</td>
<td>-5%</td>
<td>142 MB/s</td>
<td>135</td>
<td>-5%</td>
</tr>
<tr>
<td>Blowfish</td>
<td>77 MB/s</td>
<td>75</td>
<td>-2%</td>
<td>75 MB/s</td>
<td>74</td>
<td>-2%</td>
</tr>
</tbody>
</table>
Related work

- **Initial abstraction** of StealthMem (by Erlingsson and Abadi)

- **Hardware-based:**
  - Obfuscating access patterns: PLcache, RPcache ...
  - Dynamic cache partitioning
  - App. specific hardware: AES encryption instruction
    - → StealthMem works on commodity hardware

- **Software-based:**
  - Static partitioning: PageColoring
  - App. specific mitigation: reducing timing channels
    - → StealthMem provides flexible, better performance
Conclusion

• **StealthMem**: an efficient system-level protection mechanism against cache-based side channel attacks

• **Implement** the abstraction of StealthMem

• **Three** new techniques:
  • **Locking** cache lines
  • **Assigning** a private page per core
  • **Mediating** access on the private pages with PTA