Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud

Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, Kay Römer
Graz University of Technology

February 2017 — NDSS 2017
Outline

- cache covert channels
- how do we get a covert channel working in the cloud?
- how do we get a covert channel working in a noisy environment?
- what are the applications of such covert channel?
CPU cache

- main memory is slow compared to the CPU
CPU cache

- main memory is slow compared to the CPU
- caches buffer frequently used data
CPU cache

- main memory is slow compared to the CPU
- caches buffer frequently used data
- every data access goes through the cache
CPU cache

- main memory is slow compared to the CPU
- caches buffer frequently used data
- every data access goes through the cache
- caches are transparent to the OS and the software
Caches on Intel CPUs

- L1 and L2 are private

![Diagram of Intel CPU caches and slices](image.png)
Caches on Intel CPUs

- L1 and L2 are private
- Last-level cache
Caches on Intel CPUs

- L1 and L2 are private
- last-level cache
  - divided in slices
Caches on Intel CPUs

- L1 and L2 are private
- Last-level cache
  - Divided in slices
  - Shared across cores
Caches on Intel CPUs

- L1 and L2 are private
- Last-level cache
  - Divided in slices
  - Shared across cores
  - Inclusive

![Diagram of Intel CPU caches](image-url)
Caches on Intel CPUs

- L1 and L2 are private
- last-level cache
  - divided in slices
  - shared across cores
  - inclusive
- hash function maps a physical address to a slice
## Set-associative caches

<table>
<thead>
<tr>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address

Cache
Set-associative caches

Data loaded in a specific set depending on its address
Set-associative caches

Data loaded in a specific set depending on its address

Several ways per set
Set-associative caches

Data loaded in a specific set depending on its address

Several ways per set

Cache line loaded in a specific way depending on the replacement policy
Timing differences

Access time [CPU cycles]

Number of accesses

10^1 10^4 10^7

cache hits  cache misses
Cache-based covert channels

- cache attacks $\rightarrow$ exploit timing differences of memory accesses
Cache-based covert channels

- cache attacks $\rightarrow$ exploit timing differences of memory accesses
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
Cache-based covert channels

- cache attacks → exploit timing differences of memory accesses
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- literature: stops working with noise on the machine
Cache-based covert channels

- cache attacks → exploit timing differences of memory accesses
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- literature: stops working with noise on the machine
- solution? “Just use error-correcting codes”
Prime+Probe

- attacker knows which cache set the victim accessed, not the content
Prime+Probe

- attacker knows which cache set the victim accessed, not the content
- works across CPU cores as the last-level cache is shared
Prime+Probe

- attacker knows which cache set the victim accessed, not the content
- works across CPU cores as the last-level cache is shared
- does not need shared memory, e.g., memory de-deduplication
Prime+Probe

- attacker knows which cache set the victim accessed, not the content
- works across CPU cores as the last-level cache is shared
- does not need shared memory, e.g., memory de-deduplication
  → works across VM in the cloud, e.g., on Amazon EC2
Prime+Probe

Victim address space  Cache  Attacker address space
Prime + Probe

Step 1: Attacker primes, i.e., fills, the cache (no shared memory)
Prime+Probe

**Step 1:** Attacker primes, *i.e.*, fills, the cache (no shared memory)

**Step 2:** Victim evicts cache lines while running
Prime+Probe

Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running
Prime+Probe

Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed
Prime+Probe

Step 1: Attacker primes, i.e., fills, the cache (no shared memory)

Step 2: Victim evicts cache lines while running

Step 3: Attacker probes data to determine if set has been accessed
Why can’t we just use error correcting codes?

Sender 1 0 0 1 1 0
Receiver 1 0 0 1 1 0

(a) Transmission without errors
Why can’t we just use error correcting codes?

(a) Transmission without errors

Sender: 1 | 0 | 0 | 1 | 1 | 0
Receiver: 1 | 0 | 0 | 1 | 1 | 0

(b) Noise: substitution error

Sender: 1 | 0 | 0 | 1 | 1 | 0
Receiver: 1 | 1 | 0 | 1 | 1 | 0
Why can’t we just use error correcting codes?

(a) Transmission without errors

Sender: 100110
Receiver: 100110

(b) Noise: substitution error

Sender: 100110
Receiver: 110110

(c) Sender descheduled: insertions

Sender: 100110
Receiver: 100000110

(d) Receiver descheduled: deletions

Sender: 100110
Receiver: 100000110
Why can’t we just use error correcting codes?

(a) Transmission without errors

Sender 100110
Receiver 100110

(b) Noise: substitution error

Sender 100110
Receiver 110110

(c) Sender descheduled: insertions

Sender 100110
Receiver 100000110

(d) Receiver descheduled: deletions

Sender 100110
Receiver 1000000
Our robust covert channel

- **physical** layer:
  - transmits words as a sequence of ‘0’s and ‘1’s
  - deals with synchronization errors

- **data-link** layer:
  - divides data to transmit into packets
  - corrects the remaining errors
Physical layer: Sending ‘0’s and ‘1’s

- sender and receiver agree on one set
Physical layer: Sending ‘0’ s and ‘1’ s

- sender and receiver agree on one set
- receiver probes the set continuously
Physical layer: Sending ‘0’ s and ‘1’ s

- sender and receiver agree on one set
- receiver probes the set continuously
- sender transmits ‘0’ doing nothing
  → lines of the receiver still in cache → fast access
Physical layer: Sending ‘0’s and ‘1’s

- sender and receiver agree on one set
- receiver probes the set continuously
- sender **transmits ’0’** doing nothing
  - → lines of the receiver still in cache → **fast access**
- sender **transmits ’1’** accessing addresses in the set
  - → evicts lines of the receiver → **slow access**
Eviction set generation

- need a set of addresses in the **same cache set and same slice**
Eviction set generation

- need a set of addresses in the **same cache set** and **same slice**
- problem: slice number depends on all bits of the physical address
Eviction set generation

- need a set of addresses in the same cache set and same slice
- problem: slice number depends on all bits of the physical address

we can build a set of addresses in the same cache set and same slice
Eviction set generation

- need a set of addresses in the **same cache set and same slice**
- problem: slice number depends on all bits of the physical address

We can build a set of addresses in the **same cache set and same slice** without knowing **which slice**
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets
Jamming agreement

sender eviction sets

#1
#2
#3
#4

prime

Cache Sets

receiver eviction sets

_CACHE_S_ _S_ _S_ _S_ _S_ _S_ _S_ _S_ _S_

CACHE_S_ _S_ _S_ _S_ _S_ _S_ _S_ _S_ _S_
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

prime
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

probe
Jamming agreement

Cache Sets

sender eviction sets

receiver eviction sets

probe
Jamming agreement

sender eviction sets

#1 #2 #3 #4

prime

Cache Sets

receiver eviction sets

#1 #2 #3 #4
Jamming agreement

sender eviction sets

#1
#2
#3
#4

receiver eviction sets

Cache Sets

S S S S S S S S S
R R R R R R R R R

prime
Jamming agreement

Cache Sets

sender eviction sets

receiver eviction sets

probe
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

probe

S S S S S S S S S
R R R R R R R R R

/numbersign.osf/one.osf/
/numbersign.osf/two.osf/
/numbersign.osf/three.osf/
/numbersign.osf/four.osf/
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

prime
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

prime
Jamming agreement

sender eviction sets

probe

Cache Sets

receiver eviction sets

#1

#2

#3

#4
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

probe
Jamming agreement

sender eviction sets

#1
#2
#3
#4

prime

Cache Sets

receiver eviction sets

S S S S S S S S S

/numbersign.osf/one.osf
/numbersign.osf/two.osf
/numbersign.osf/three.osf
/numbersign.osf/four.osf

Cache Sets

14 / 25
Jamming agreement

sender eviction sets

#1
#2
#3
#4

receiver eviction sets

Cache Sets

prime

/one.osf /four.osf / two.osf /five.osf
Jamming agreement

sender eviction sets

probe

receiver eviction sets

Cache Sets

S  S  S  S  S  S  S  S  S
Jamming agreement

sender eviction sets

receiver eviction sets

Cache Sets

probe

#1
Jamming agreement

sender eviction sets

#1
#2
#3
#4

receiver eviction sets

Cache Sets

#1
Jamming agreement

sender eviction sets

#1
#2
#3
#4

Cache Sets

receiver eviction sets

#1
Jamming agreement

sender eviction sets

#1 ✓
#2
#3
#4

receiver eviction sets

#1

repeat!
Jamming agreement

sender eviction sets

#1
#2
#3
#4

receiver eviction sets

#1
#2

repeat!
Jamming agreement

sender eviction sets

#1
#2
#3
#4

receiver eviction sets

#1
#2
#3

repeat!
Jamming agreement

sender eviction sets

receiver eviction sets

repeat!
Sending the first image
Handling synchronization errors

Physical layer word

<table>
<thead>
<tr>
<th>Data</th>
<th>12 bits</th>
</tr>
</thead>
</table>

Data

12 bits
Handling synchronization errors

- deletion errors: request-to-send scheme that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)

Physical layer word

- Data: 12 bits
- SQN: 3 bits
Handling synchronization errors

- deletion errors: request-to-send scheme that also serves as ack
  - 3-bit sequence number
  - request: encoded sequence number (7 bits)
- ’o’-insertion errors: error detection code → Berger codes
  - appending the number of ’o’s in the word to itself
  → property: a word cannot consist solely of ’o’s

Physical layer word

<table>
<thead>
<tr>
<th>Data</th>
<th>SQN</th>
<th>EDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bits</td>
<td>3 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>
Synchronization (before)
Synchronization (after)
Synchronization (after)
Synchronization (after)
Data-link layer: Error correction

- **Reed-Solomon** codes to correct the remaining errors
Data-link layer: Error correction

- **Reed-Solomon** codes to correct the remaining errors
- RS word size = physical layer word size = 12 bits
- packet size = $2^{12} - 1 = 4095$ RS words
- 10% error-correcting code: 409 parity and 3686 data RS words
Error correction (after)
## Evaluation

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
</tbody>
</table>
## Evaluation

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
</tbody>
</table>
# Evaluation

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>-</td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.25 KBps</td>
<td>0.00%</td>
<td>-</td>
</tr>
</tbody>
</table>
# Evaluation

<table>
<thead>
<tr>
<th>Environment</th>
<th>Bit rate</th>
<th>Error rate</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>75.10 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Native</td>
<td>36.03 KBps</td>
<td>0.00%</td>
<td>stress -m 1</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.25 KBps</td>
<td>0.00%</td>
<td>–</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>45.09 KBps</td>
<td>0.00%</td>
<td>web server serving files on sender VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>42.96 KBps</td>
<td>0.00%</td>
<td>stress -m 2 on sender VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>42.26 KBps</td>
<td>0.00%</td>
<td>stress -m 1 on receiver VM</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>37.42 KBps</td>
<td>0.00%</td>
<td>web server on all 3 VMs, stress -m 4 on 3rd VM, stress -m 1 on sender and receiver VMs</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>34.27 KBps</td>
<td>0.00%</td>
<td>stress -m 8 on third VM</td>
</tr>
</tbody>
</table>
Building an SSH connection

VM 1

TCP Client
(e.g. ssh)

TCP ↔ File

File System

Covert Channel

Socket

Hypervisor

Prime+Probe

Last Level Cache (LLC)

VM 2

TCP Server
(e.g. sshd)

TCP ↔ File

File System

Covert Channel

Socket

Prime+Probe
SSH evaluation

Between two instances on Amazon EC2

<table>
<thead>
<tr>
<th>Noise</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No noise</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 8</code> on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on SSH server VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on all VMs</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 1</code> on server side</td>
<td>unstable</td>
</tr>
</tbody>
</table>
### SSH evaluation

Between two instances on Amazon EC2

<table>
<thead>
<tr>
<th>Noise</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No noise</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 8 on third VM</code></td>
<td>✓</td>
</tr>
<tr>
<td>Web server on third VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on SSH server VM</td>
<td>✓</td>
</tr>
<tr>
<td>Web server on all VMs</td>
<td>✓</td>
</tr>
<tr>
<td><code>stress -m 1 on server side</code></td>
<td>unstable</td>
</tr>
</tbody>
</table>

Telnet also works with occasional corrupted bytes with `stress -m 1`
Conclusion

- cache covert channels are practical
Conclusion

- cache covert channels are **practical**
- even in the **cloud**, even in presence of extraordinary **noise**
Conclusion

- cache covert channels are practical
- even in the cloud, even in presence of extraordinary noise
- our robust covert channel supports an SSH connection
Conclusion

- cache covert channels are practical
- even in the cloud, even in presence of extraordinary noise
- our robust covert channel supports an SSH connection
- we extended Amazon’s product portfolio :)

/two.osf/four.osf / /two.osf/five.osf
Conclusion

- cache covert channels are practical
- even in the cloud, even in presence of extraordinary noise
- our robust covert channel supports an SSH connection
- we extended Amazon’s product portfolio :)

![Amazon Prime Logo](https://www.amazon.com)
Conclusion

- cache covert channels are practical
- even in the cloud, even in presence of extraordinary noise
- our robust covert channel supports an SSH connection
- we extended Amazon’s product portfolio :)
Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud

Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, Kay Römer Grazi University of Technology

February 2017 — NDSS 2017