Safecracker: Leaking Secrets through Compressed Caches

Po-An Tsai, Andres Sanchez, Christopher Fletcher, and Daniel Sanchez

ASPLOS 2020
Executive Summary

- First security analysis of cache compression
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets

Attacker

Victim
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets

Attacker sends encryption request to victim
Victim stores input next to key
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets

Attacker

1. Attacker sends encryption request to victim

Victim

2. Victim stores input next to key

encrypt 0x01...

Victim stores input next to key
0x01020304050607 0x01

Cache compresses line
7B cache line
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets

**Diagram:**

1. **Attacker** sends encryption request to **victim**
2. **Victim** stores input next to key
3. **Attacker** measures line’s compressed size, infers 
   
   ![Diagram](image-url)
Executive Summary

- First security analysis of cache compression
- Compressibility of a cache line reveals info about its data
- Attacker can exploit data colocation to leak secrets

Compromises secret key in ~10ms
First security analysis of cache compression

Compressibility of a cache line reveals info about its data

Attacker can exploit data colocation to leak secrets

Compromises secret key in ~10ms
Leaks large fraction of victim memory when combined latent memory safety vulnerabilities
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Kiriansky et. al, MICRO’18
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

Victim’s protection domain

Secret

Transmitter

Receiver

Attacker’s protection domain

Secret

Kiriansky et al, MICRO’18
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

Presence of a line and its address (location in cache)

Victim’s protection domain

Attacker’s protection domain

Kiriansky et. al, MICRO’18
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

- Speculatively executed instructions
- Presence of a line and its address (location in cache)

Kiriansky et. al, MICRO’18
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

Speculatively executed instructions
Presence of a line and its address (location in cache)
Timing difference to infer a line’s presence

Victim’s protection domain
Secret → Transmitter → Receiver → Secret

Side channel

Attacker’s protection domain
Kiriansky et al, MICRO’18
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)
- Speculatively executed instructions
- Presence of a line and its address (location in cache)
- Timing difference to infer a line’s presence

Victim’s protection domain
- Secret
- Transmitter
- Side channel

Attacker’s protection domain
- Receiver
- Secret

Kiriansky et al., MICRO’18

Compressed cache attacks
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

- Speculatively executed instructions
- Presence of a line and its address (location in cache)
- Timing difference to infer a line’s presence

Victim’s protection domain

Secret

Transmitter

Side channel

Kiriansky et. al, MICRO’18

Receiver

Attacker’s protection domain

Secret

Compressibility of secret (and data in same line)

Compressed cache attacks
Speculation-Based vs. Compressed Cache Side-Channel Attacks

- Speculation-based cache side channel attacks (e.g., Spectre)
  - Speculatively executed instructions
  - Presence of a line and its address (location in cache)
  - Timing difference to infer a line’s presence

Victim’s protection domain
- Secret
- Transmitter
- Side channel

Attacker’s protection domain
- Receiver
- Secret

Kiriansky et. al, MICRO’18

Writing secret data (or data in same line)
Compressibility of secret (and data in same line)
Compressed cache attacks
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Speculation-based cache side channel attacks (e.g., Spectre)

Speculatively executed instructions
Presence of a line and its address (location in cache)
Timing difference to infer a line’s presence

Victim’s protection domain

Secret → Transmitter → Receiver → Secret

Side channel

Attacker’s protection domain

Kiriansky et. al, MICRO’18

Compressibility of secret (and data in same line)
Timing difference to infer a line’s compressibility

Writing secret data (or data in same line)

Compressed cache attacks

Compressibility of secret (and data in same line)
Timing difference to infer a line’s presence
Speculation-Based vs. Compressed Cache Side-Channel Attacks

Kiriansky et. al, MICRO’18

Compressed cache attacks leak data without relying on speculation
Outline

- Background on cache compression
- Pack+Probe: Measuring cache line compressibility
- Safecracker: Exploiting data colocation to leak secrets
- Potential defenses
Cache Compression Tradeoffs

- Higher effective capacity → Higher hit rate
- Somewhat higher hit latency
Cache Compression Tradeoffs

- Higher effective capacity $\rightarrow$ Higher hit rate
- Somewhat higher hit latency
- Highly beneficial for large caches (e.g., LLC)
Cache Compression Tradeoffs

- Higher effective capacity $\rightarrow$ Higher hit rate
- Somewhat higher hit latency
- Highly beneficial for large caches (e.g., LLC)
- Intense research activity over past 15 years
Cache Compression Tradeoffs

- Higher effective capacity $\rightarrow$ Higher hit rate
- Somewhat higher hit latency
- Highly beneficial for large caches (e.g., LLC)
- Intense research activity over past 15 years

A Case for Toggle-Aware Compression for GPU Systems

Gennady Pekhimenko¹, Evgeny Bolotin², Naradita Vijaykumar¹, Omar Mutlu¹, Todd C. Mowry¹, Stephen W. Keckler¹²

¹Carnegie Mellon University, ²NVIDIA, ³University of Texas at Austin

ABSTRACT

Data compression can be an effective method to achieve higher system performance and energy efficiency in modern data-intensive applications by exploiting redundancy and data similarity. Prior works have studied a variety of data compression techniques to improve both capacity (e.g., of caches and main memory) and bandwidth utilization (e.g., of the on-chip and off-chip interconnects). In this paper, we make a new observation about the energy efficiency of communication when compression is applied. While compression reduces the amount of transferred data, it leads to a substantial increase in the number of bit toggles (i.e., communication channel switchings) from 0 to 1 or from 1 to 0. The increased toggle count increases the dynamic energy consumed by on-chip and off-chip buses due to state transition charging and discharging of the wires. Our bandwidth utilization (e.g., of on-chip and off-chip interconnects [15, 3, 64, 58, 51, 60, 69]) and GPUs [58, 51, 69], which results in better system performance and energy consumption. Bandwidth compression proves to be particularly effective in GPUs because they are often bottlenecked by memory bandwidth [47, 32, 31, 72, 69]. GPU applications also exhibit high degrees of data redundancy [58, 51, 69], leading to good compression ratios.

While data compression can dramatically reduce the number of bit symbols that must be transmitted across a link, compression also carries two well-known overheads: (1) latency, energy, and area overhead of the compression/decompression hardware [4, 52]; and (2) complexity and cost to implement.
Cache Compression Tradeoffs

- Higher effective capacity \(\rightarrow\) Higher hit rate
- Somewhat higher hit latency
- Highly beneficial for large caches (e.g., LLC)
- Intense research activity over past 15 years

All focus on performance, not security
Cache Compression Ingredients
Cache Compression Ingredients

- Architecture: How to locate and manage variable-sized compressed blocks?
Cache Compression Ingredients

- Architecture: How to locate and manage variable-sized compressed blocks?
- Algorithm: How to compress each cache block?
Cache Compression Ingredients

- **Architecture**: How to locate and manage variable-sized compressed blocks?
- **Algorithm**: How to compress each cache block?
- We focus attacks on a commonly used baseline:
  - VSC compressed cache architecture
  - BDI compression algorithm
Cache Compression Ingredients

- **Architecture:** How to locate and manage variable-sized compressed blocks?
- **Algorithm:** How to compress each cache block?
- We focus attacks on a commonly used baseline:
  - VSC compressed cache architecture
  - BDI compression algorithm
- Attacks apply to other architectures & algorithms
  - Leads to different characteristics about leaked data
Conventional caches can only manage fixed-size blocks
VSC divides data array into small segments and lets compressed lines take a variable number of segments.
VSC divides data array into small segments and lets compressed lines take a variable number of segments.
VSC divides data array into small segments and lets compressed lines take a variable number of segments.
VSC [Alameldeen and Wood ISCA‘04]

- VSC divides data array into small segments and lets compressed lines take a variable number of segments.
- VSC increases tags relative to uncompressed caches to track more compressed lines per set.
- VSC divides data array into small segments and lets compressed lines take a variable number of segments.
- VSC increases tags relative to uncompressed caches to track more compressed lines per set.
- VSC divides data array into small segments and lets compressed lines take a variable number of segments.
- VSC increases tags relative to uncompressed caches to track more compressed lines per set.
VSC [Alameldeen and Wood ISCA‘04]

- VSC divides data array into small segments and lets compressed lines take a variable number of segments.
- VSC increases tags relative to uncompressed caches to track more compressed lines per set.
Base-Delta-Immediate (BDI) compresses lines with similar values by using a common base + small deltas.

![Diagram of BDI compression]

32-byte Uncompressed Cache Line

12-byte Compressed Cache Line

Saved Space
Base-Delta-Immediate (BDI) compresses lines with similar values by using a common base + small deltas.

BDI supports multiple formats with different base sizes (2, 4, 8 bytes) and delta sizes (1, 2, 4 bytes).
- **Base-Delta-Immediate (BDI)** compresses lines with similar values by using a common base + small deltas

- BDI supports multiple formats with different base sizes (2, 4, 8 bytes) and delta sizes (1, 2, 4 bytes)

- Reasonable compression ratio, simple implementation
Threat model:
- Attacker and victim run in different protection domains (processes, VMs, etc.)
- Attacker and victim share compressed cache
- Attacker knows compressed cache architecture & algorithm used
- Attacker knows set of victim’s target line (can use standard techniques to find it)
Pack+Probe: Measuring Compressibility

- Threat model:
  - Attacker and victim run in different protection domains (processes, VMs, etc.)
  - Attacker and victim share compressed cache
  - Attacker knows compressed cache architecture & algorithm used
  - Attacker knows set of victim’s target line (can use standard techniques to find it)

- Goal: Find compressed size of target line
Attacker packs target set with lines of known sizes, leaving $S$ free segments and at least one free tag.
Attacker packs target set with lines of known sizes, leaving $S$ free segments and at least one free tag.
Attacker **packs** target set with lines of known sizes, leaving $S$ free segments and at least one free tag.
Attackers pack target set with lines of known sizes, leaving $S$ free segments and at least one free tag.

After victim accesses target set, attackers probe all lines used to pack target set:

- All hits $\rightarrow$ Victim line $\leq S$ segments
- Any miss $\rightarrow$ Victim line $> S$ segments
Pack+Probe: Measuring Compressibility

Attacker **packs** target set with lines of known sizes, leaving $S$ free segments and at least one free tag.

After victim accesses target set, attacker **probes** all lines used to pack target set:

- All hits $\rightarrow$ Victim line $\leq S$ segments
- Any miss $\rightarrow$ Victim line $> S$ segments
Attacker packs target set with lines of known sizes, leaving $S$ free segments and at least one free tag.

After victim accesses target set, attacker probes all lines used to pack target set:

- All hits $\rightarrow$ Victim line $\leq S$ segments
- Any miss $\rightarrow$ Victim line $> S$ segments

$S=4$
Attacker packs target set with lines of known sizes, leaving $S$ free segments and at least one free tag.

After victim accesses target set, attacker probes all lines used to pack target set:

- All hits $\Rightarrow$ Victim line $\leq S$ segments
- Any miss $\Rightarrow$ Victim line $> S$ segments

By doing a binary search over $S$, one can find exact size in $\log_2(\text{MaxSegmentsPerCacheLine})$ measurements.
Threat model:
- Attacker and victim run in different domains, share compressed cache (as in Pack+Probe)
- Attacker can get victim to collocate attacker-controlled data near victim’s own secret data

Goal: Leak victim’s data
Safecracker: Exploiting Data Colocation to Leak Secrets

- **Threat model:**
  - Attacker and victim run in different domains, share compressed cache (as in Pack+Probe)
  - Attacker can get victim to collocate attacker-controlled data near victim’s own secret data

- **Goal:** Leak victim’s data

- **Multiple colocation vectors:**
  - Victim itself collocates (contiguous allocation, stack spills, etc.)
  - Memory safety violations (buffer overflows, heap spraying, etc.)
Safecracker: Exploiting Data Colocation to Leak Secrets

- **Threat model:**
  - Attacker and victim run in different domains, share compressed cache (as in Pack+Probe)
  - Attacker can get victim to collocate attacker-controlled data near victim’s own secret data

- **Goal:** Leak victim’s data

- **Multiple colocation vectors:**
  - Victim itself colocates (contiguous allocation, stack spills, etc.)
  - Memory safety violations (buffer overflows, heap spraying, etc.)

- **Safecracker changes attacker-controlled data to reveal nearby secret data through changes in compressibility**
  - Search strategy depends on compression algorithm
Starting from largest delta, sweep high-order bytes until target line decreases in size
Safecracker on BDI

- Starting from largest delta, sweep high-order bytes until target line decreases in size

```
0x00000000 ... 0x00000000
```

32-byte Uncompressed Cache Line

- Attacker-controlled input
- Secret data

Compressed size: 32B
Safecracker on BDI

- Starting from largest delta, sweep high-order bytes until target line decreases in size.

<table>
<thead>
<tr>
<th>Attacker-controlled input</th>
<th>Secret data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x0F00BA20</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x0F00BA20</td>
</tr>
</tbody>
</table>

32B compressed size

- Secret data
- Attacker-controlled input
Starting from largest delta, sweep high-order bytes until target line decreases in size

<table>
<thead>
<tr>
<th>4 bytes</th>
<th>4 bytes</th>
<th>32-byte Uncompressed Cache Line</th>
<th>Compressed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>...</td>
<td>0x00000000 0xF00BA20</td>
<td>32B</td>
</tr>
<tr>
<td>0x00010000</td>
<td>...</td>
<td>0x00010000 0xF00BA20</td>
<td>32B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32B</td>
</tr>
<tr>
<td>0xF0000000</td>
<td>...</td>
<td>0xF0000000 0xF00BA20</td>
<td>32B</td>
</tr>
</tbody>
</table>

Attacker-controlled input

Secret data
Safecracker on BDI

- Starting from largest delta, sweep high-order bytes until target line decreases in size

<table>
<thead>
<tr>
<th>4 bytes</th>
<th>4 bytes</th>
<th>32-byte Uncompressed Cache Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>...</td>
<td>0x00000000 0xF00BA20</td>
</tr>
<tr>
<td>0x00010000</td>
<td>...</td>
<td>0x00010000 0xF00BA20</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xF0000000</td>
<td>...</td>
<td>0xF000000 0xF00BA20</td>
</tr>
</tbody>
</table>

Compressed size: 32B

4B base  2B deltas

| 0xF0000000 0000 0000 0000 0000 0000 0000 0000 | BA20 |

20B!
Continue sweeping lower-order bytes until recovering all bytes
Continue sweeping lower-order bytes until recovering all bytes.
Safecracker on BDI

- Continue sweeping lower-order bytes until recovering all bytes

```
0x0F000100 ... 0x0F000100 0x0F00BA20

... 0x0F00BA00 0x0F00BA00
```

**Compressed size**
- 20B
- 12B

- Attacker-controlled input
- Secret data
Continue sweeping lower-order bytes until recovering all bytes

32-byte Uncompressed Cache Line

```
0x0F000100 ... 0x0F000100 0x0F00BA20
... 0xF00BA00 0x0F00BA20
... 0xF00BA20 0x0F00BA20
```

Compressed size

- 20B
- 12B
- 8B

Attacker-controlled input
Secret data
Safecracker on BDI

- Continue sweeping lower-order bytes until recovering all bytes
  - 32-byte Uncompressed Cache Line
    - Attacker-controlled input
    - Secret data
    - Compressed size
    - 20B
    - 12B
    - 8B

- BDI allows recovering up to 8 bytes this way

<table>
<thead>
<tr>
<th>Secret Size</th>
<th>Compression Format Sequence</th>
<th>Attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2B</td>
<td>NoComp→B2D1→B8D0</td>
<td>O(2^8)</td>
</tr>
<tr>
<td>4B</td>
<td>NoComp→B4D2→B4D1→B8D0</td>
<td>O(2^16)</td>
</tr>
<tr>
<td>8B</td>
<td>NoComp→B8D4→B8D2→B8D1→B8D0</td>
<td>O(2^32)</td>
</tr>
</tbody>
</table>
Buffer overflows let Safecracker control where attacker-controlled data is located

- Makes search more efficient
- Can leak data far away from buffer
Buffer overflows let Safecracker control where attacker-controlled data is located

- Makes search more efficient
- Can leak data far away from buffer

With BDI, can leak 1/8\textsuperscript{th} of victim’s memory!

- Other compression algorithms (e.g., RLE) allow more leakage
Safecracker Evaluation

- Microarchitectural simulation using zsim
- Multicore system modeled after Skylake
Safecracker Evaluation

- Microarchitectural simulation using zsim
- Multicore system modeled after Skylake
- Two Proof-of-Concept (PoC) workloads:
  - Login server that collocates key and attacker data
  - Server with buffer overflow + key elsewhere in stack

Diagram:

- Main Memory
- Compressed LLC: 8MB VSC with 64-byte lines, 2x tag array, 32 tags/set
- BDI compression
- Core
- L2
- L2
- Core
Safecracker steals secrets quickly

PoC 1: Fixed colocation

![Graph showing execution time vs. size of the secret to recover (Bytes)]
Safecracker steals secrets quickly

Leaks 4B in under 100ms, 6B in 200ms (comparable to time spent finding target set)
Safecracker steals secrets quickly

Leaks 4B in under 100ms, 6B in 200ms (comparable to time spent finding target set)

8B would take much longer (~90 hours)
Safecracker steals secrets quickly

PoC 1: Fixed colocation

PoC 2: Buffer overflow

Leaks 4B in under 100ms, 6B in 200ms (comparable to time spent finding target set)

8B would take much longer (~90 hours)
Safecracker steals secrets quickly

**PoC 1: Fixed colocation**

Leaks 4B in under 100ms, 6B in 200ms (comparable to time spent finding target set)

Leaks 8B would take much longer (~90 hours)

**PoC 2: Buffer overflow**

Leaks 8B in ~10ms

Attack time grows linearly with leaked bytes
Most compressed cache architectures allow conflicts among a small set of lines. Pack+Probe still applies.
Generalizing attacks to other compressed caches

- Most compressed cache architectures allow conflicts among a small set of lines → Pack+Probe still applies
  - See paper for more discussions
Most compressed cache architectures allow conflicts among a small set of lines $\rightarrow$ Pack+Probe still applies

- See paper for more discussions

Compressibility *always* leaks information about data

- More info the better the compression algorithm is
Generalizing attacks to other compressed caches

- Most compressed cache architectures allow conflicts among a small set of lines → Pack+Probe still applies
  - See paper for more discussions

- Compressibility always leaks information about data
  - More info the better the compression algorithm is
  - Adaptive compression algorithms use shared state
Generalizing attacks to other compressed caches

- Most compressed cache architectures allow conflicts among a small set of lines → Pack+Probe still applies
  - See paper for more discussions

- Compressibility always leaks information about data
  - More info the better the compression algorithm is
  - Adaptive compression algorithms use shared state
    → additional attack vector
Defense against cache compression attacks
Defense against cache compression attacks

- Cache partitioning for isolation
  - Prevents attacks without software changes
  - Invasive: must partition both tag and data arrays
Defense against cache compression attacks

- Cache partitioning for isolation
  - Prevents attacks without software changes
  - Invasive: must partition both tag and data arrays

- Performance distribution of 25 mixes of 4 SPEC CPU2006 apps, using no and static partitioning:
Defense against cache compression attacks

- Cache partitioning for isolation
  - Prevents attacks without software changes
  - **Invasive**: must partition both tag and data arrays

- Performance distribution of 25 mixes of 4 SPEC CPU2006 apps, using no and static partitioning:

  Partitioning increases fragmentation in VSC, reduces effective compression ratio
Other possible defenses for compressed cache attacks

Examples of vulnerable apps due to colocation with attacker-controlled data

Discussion on generalizing attacks to other compressed caches

Artifact description
Compressed caches introduce new side channel & attacks
Conclusions

- Compressed caches introduce new side channel & attacks

- Pack+Probe exploits compressed cache architectures to observe compressibility of victim’s lines
Conclusions

- Compressed caches introduce new side channel & attacks

- Pack+Probe exploits compressed cache architectures to observe compressibility of victim’s lines

- Safecracker exploits compression algorithms + colocation of attacker-controlled & secret data to leak data quickly
  - Can leak a large fraction of program data
  - Potentially as damaging as speculation-based attacks
Conclusions

- Compressed caches introduce new side channel & attacks

- Pack+Probe exploits compressed cache architectures to observe compressibility of victim’s lines

- Safecracker exploits compression algorithms + colocation of attacker-controlled & secret data to leak data quickly
  - Can leak a large fraction of program data
  - Potentially as damaging as speculation-based attacks

- Defenses have drawbacks
  - Motivates future work on efficient defenses
Attacker sends encryption request to victim
Attacker measures line’s compressed size, infers 0x01 is in the secret data
Compromises secret key in ~10ms

Victim stores input next to key
Cache compresses line

Safecracker: Leaking Secrets through Compressed Caches