Cache Side-Channel Attacks and the case of Rowhammer

Daniel Gruss
IAIK, Graz University of Technology

April 28, 2016

1. Caches
2. Cache attacks
3. Cache Template Attacks
4. Rowhammer

1. Caches
2. Cache attacks
3. Cache Template Attacks
4. Rowhammer

CPU Caches

- CPU speed increases
- Latency of DRAM is too high
- Problem: DRAM is a bottle neck

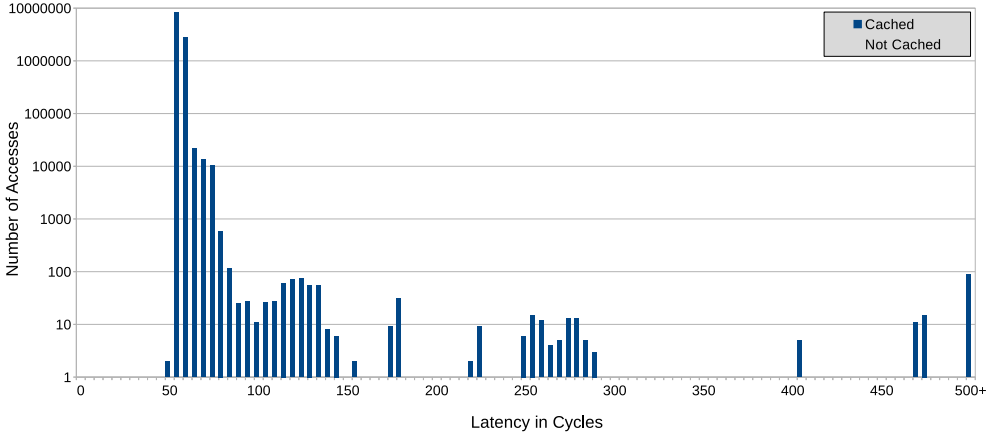
CPU Caches

- CPU speed increases
- Latency of DRAM is too high
- Problem: DRAM is a bottle neck
- Solution: Caches - fast/small memory buffers

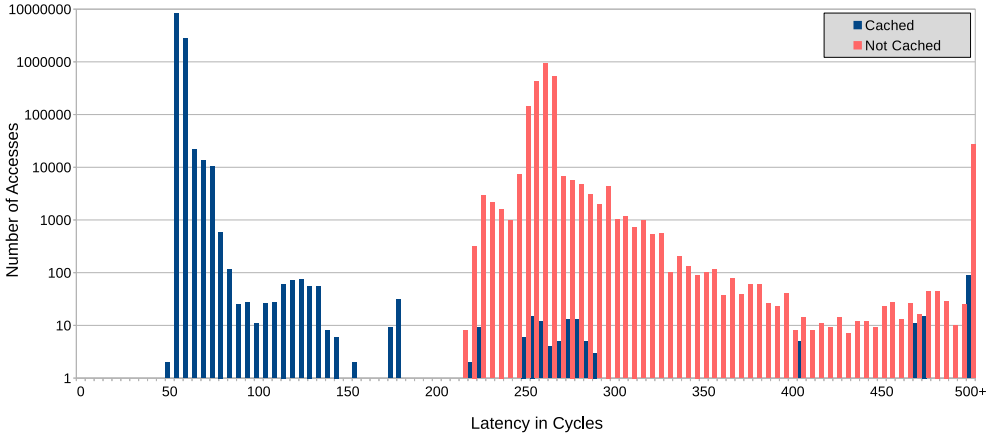
Memory accesses are cached

- Every memory reference goes through the cache
- Transparent to OS and programs

Memory Access Latency



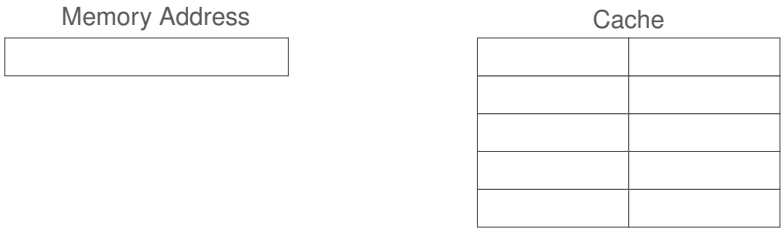
Memory Access Latency



Directly mapped cache

Memory Address

Directly mapped cache



Directly mapped cache

Memory Address

Cache

Tag	Data

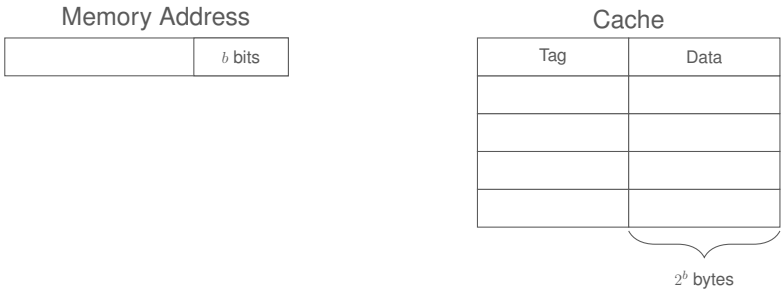
Directly mapped cache

Memory Address

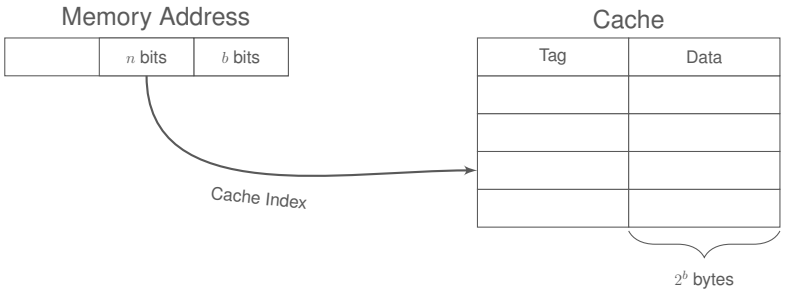
Cache

Tag	Data

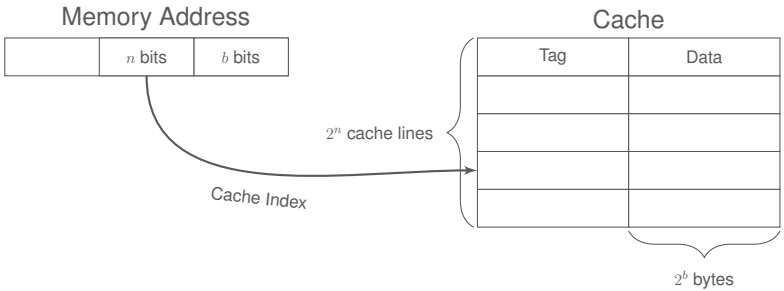
Directly mapped cache



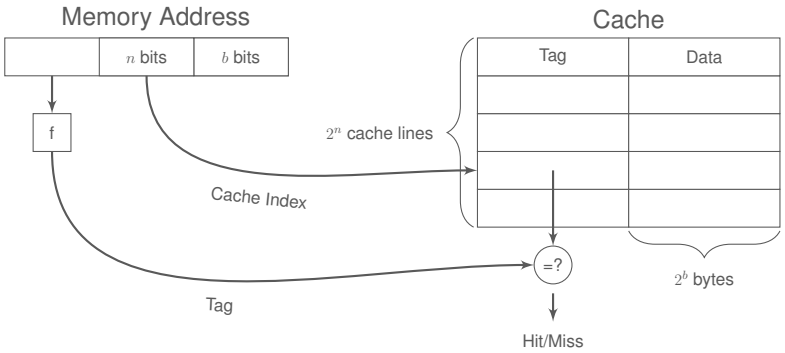
Directly mapped cache



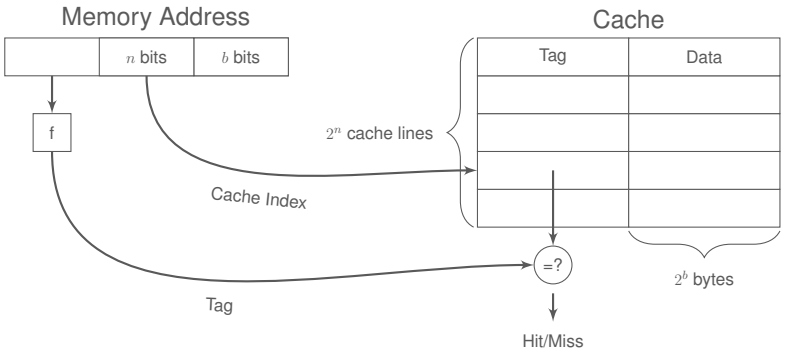
Directly mapped cache



Directly mapped cache

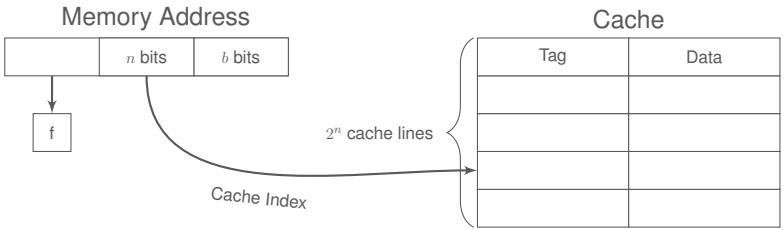


Directly mapped cache

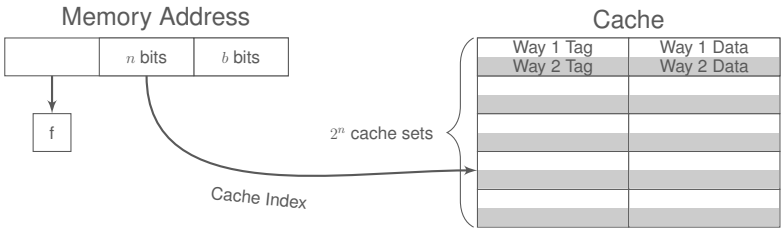


Problem: working on congruent addresses

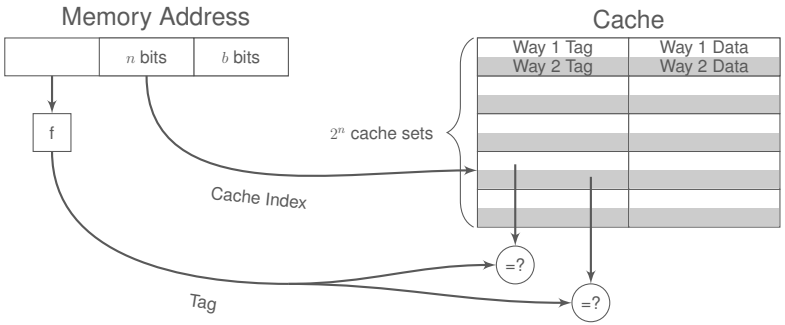
2-way set associativity



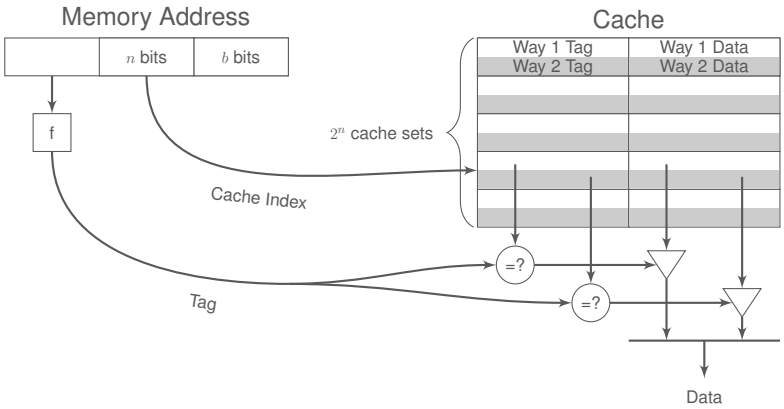
2-way set associativity



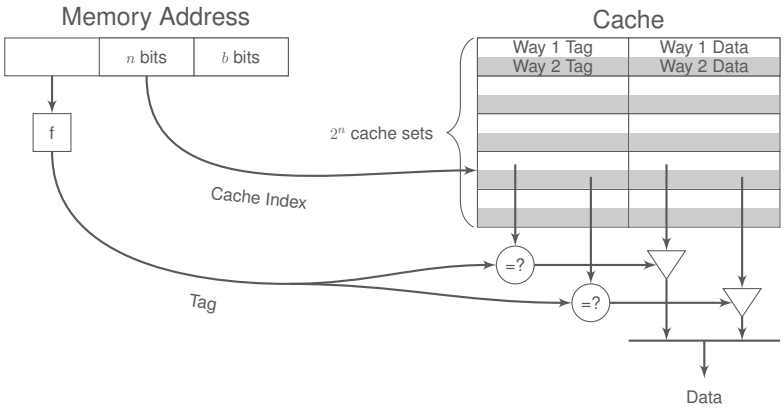
2-way set associativity



2-way set associativity



2-way set associativity



→ replacement policy is important

Caches and Addresses

- Often use physical addresses
- Virtual-to-physical translation necessary

TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency

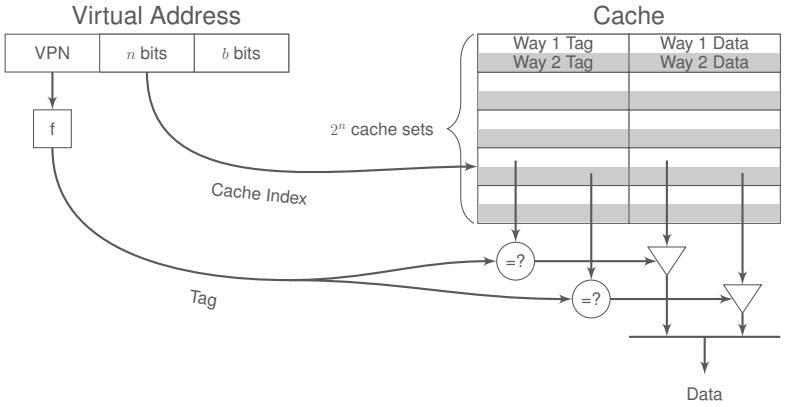
TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
- Worst case for Cache: mapping not in TLB, need to load mapping from RAM
- Solution: Use virtual addresses instead of physical addresses

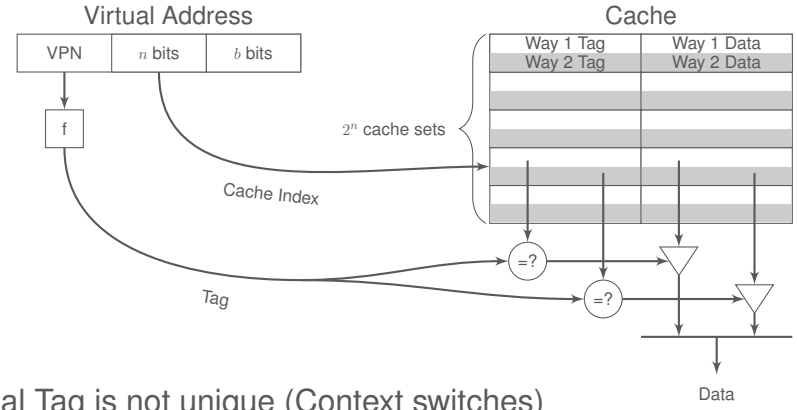
Cache indexing methods

- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged
- PIVT: Physically indexed, virtually tagged
- VIPT: Virtually indexed, physically tagged

VIVT

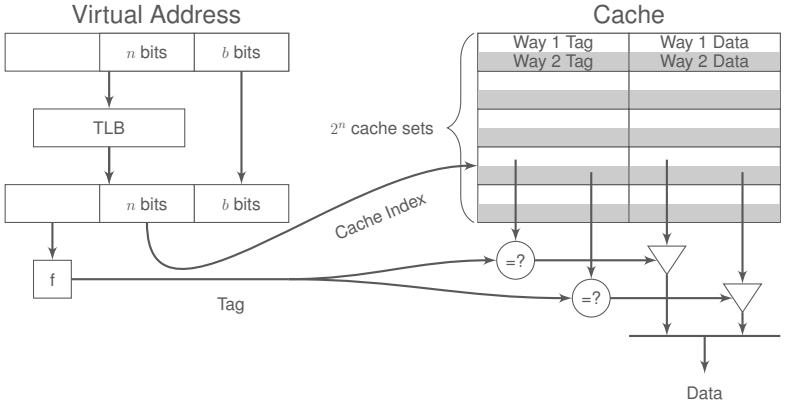

Virtual Address
VPN
n bits
b bits
f
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
2^n cache sets
Cache Index
Tag
=?
=?
Data

VIVT

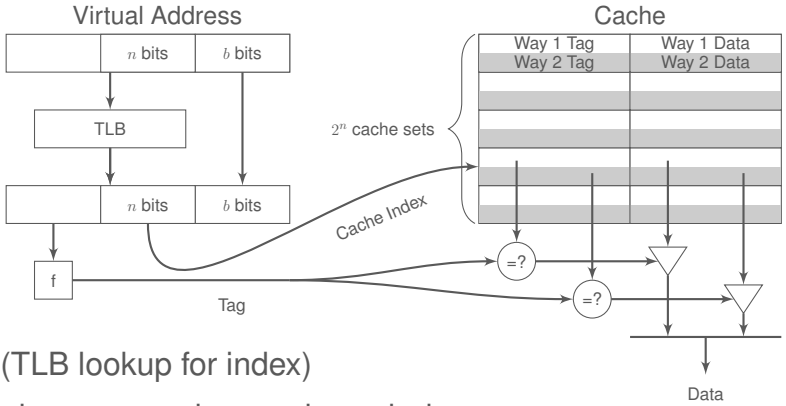


- Fast
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

PIPT

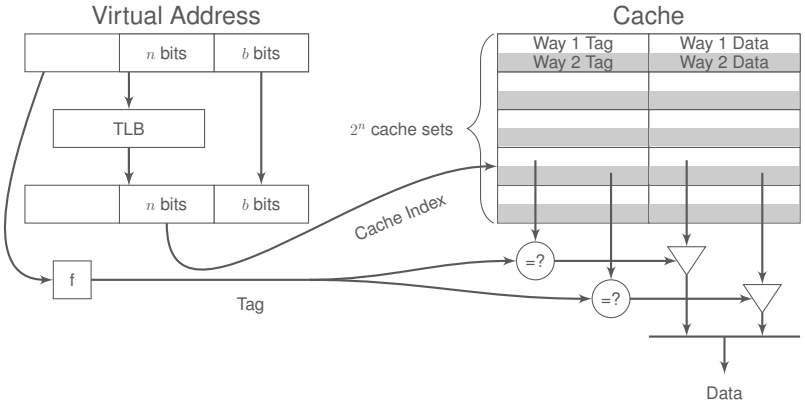

Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
2^n cache sets
=?
=?
Data

PIPT

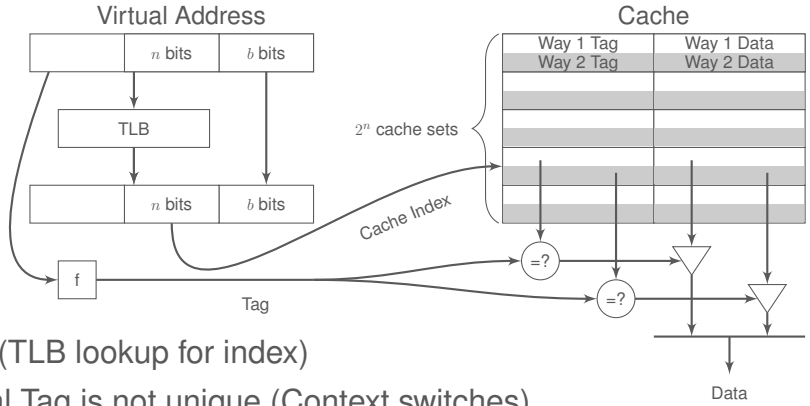


- Slow (TLB lookup for index)
- Shared memory only once in cache!

(PIVT)

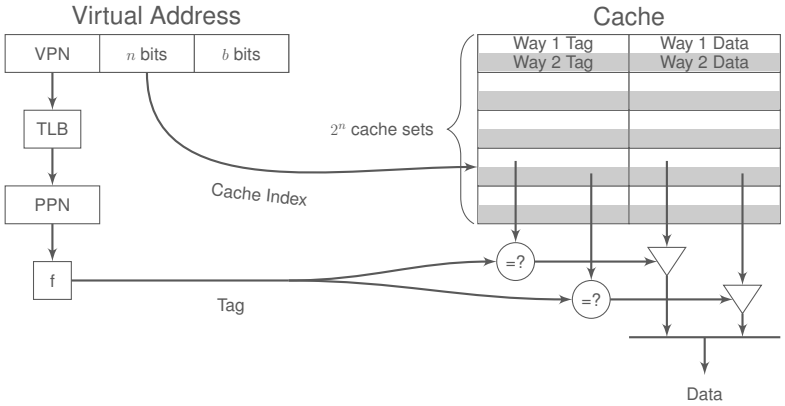

Virtual Address
n bits
b bits
TLB
n bits
b bits
f
Tag
Cache Index
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

(PIVT)

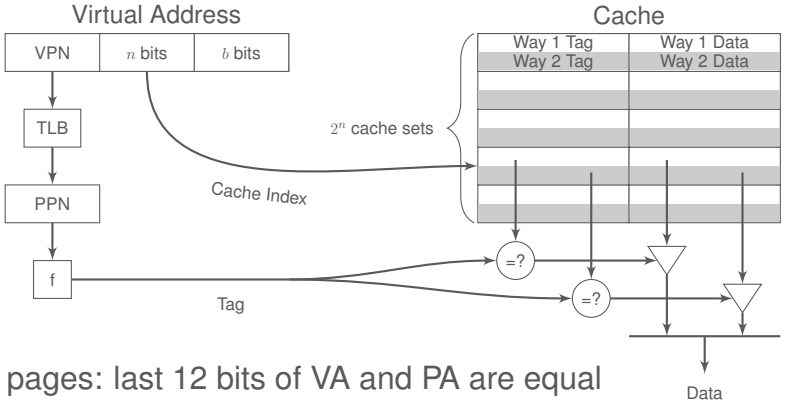


- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache

VIPT


Virtual Address
VPN
n bits
b bits
TLB
PPN
f
Cache Index
Tag
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
2^n cache sets
=?
=?
Data

VIPT



- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal
- Using more bits has disadvantages (like VIVT)

$\rightarrow$ Cache size $\leq$ # ways $\cdot$ page size

Remarks

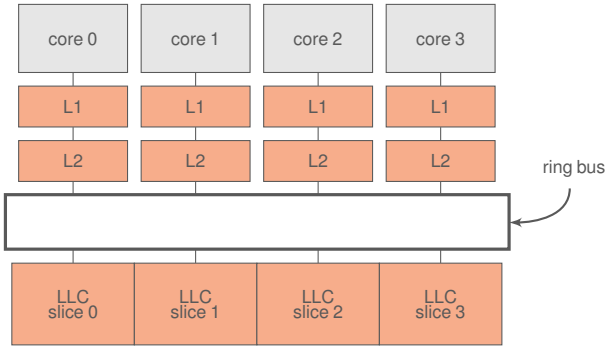
- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT

Remarks

- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT

  $\rightarrow$ shared memory is shared in cache!

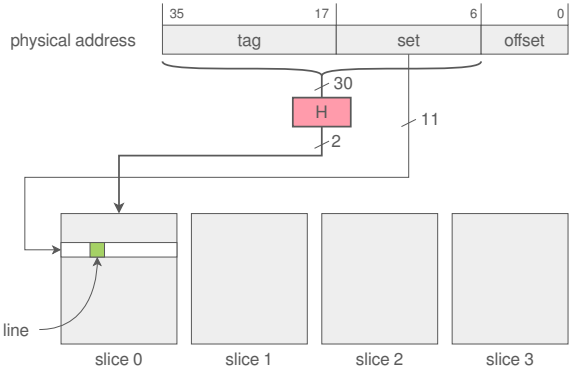
Caches today



- L1 and L2 are private
- last-level cache:
  - divided in slices
  - shared across cores
  - inclusive

  → shared memory shared is in cache, across cores!

Cache mapping



- Function H that maps slices is undocumented
- Reverse-engineered by [Hund et al., 2013, Maurice et al., 2015a, Inci et al., 2015, Yarom et al., 2015]

Unprivileged cache maintainance

User programs can optimize cache usage:

- `prefetch`: suggest CPU to load data into cache
- `clflush`: throw out data from from all caches

... based on virtual addresses

1. Caches
2. Cache attacks
3. Cache Template Attacks
4. Rowhammer

Cache attacks

- cache-based keylogging
- crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% of RSA key bits recovered after 1 encryption
- cross-VM, cross-core, even cross-CPU
- any architecture (Intel, AMD, ARM)

Cross-core attacks?

- exploiting the inclusive property

Cross-core attacks?

- exploiting the inclusive property
- last-level cache is a superset of L1 and L2

Cross-core attacks?

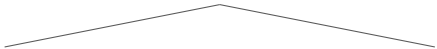
- exploiting the inclusive property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2

Cross-core attacks?

- exploiting the inclusive property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2
- a core can evict lines in the private L1 cache of another core

Access-driven attacks

Attacker monitors its own activity to determine cache lines or sets accessed by victim.



Flush+Reload

[Gullasch et al., 2011]

[Yarom and Falkner, 2014]

[Gruss et al., 2015b]

Prime+Probe

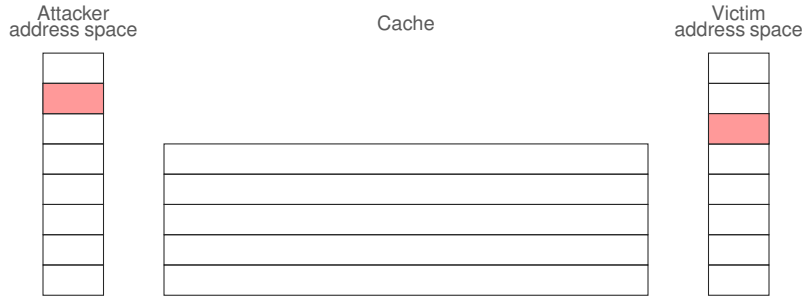
[Percival, 2005]

[Liu et al., 2015]

[Maurice et al., 2015b]

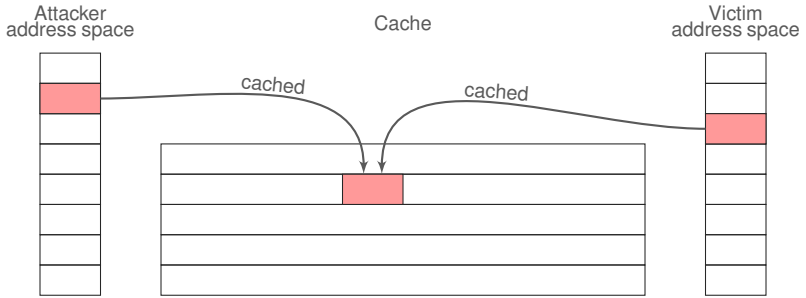
Same techniques for covert and side channels

Flush+Reload



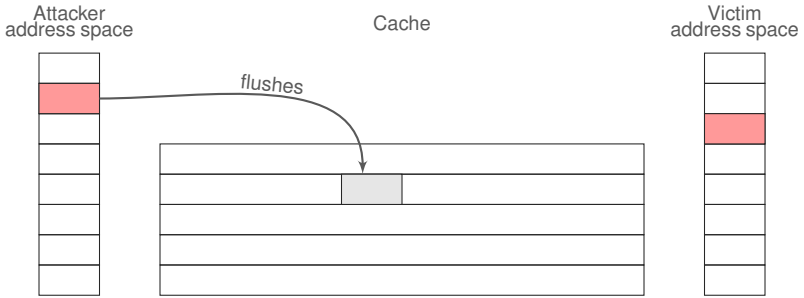
step 0: attacker maps shared library → shared memory, shared in cache

Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

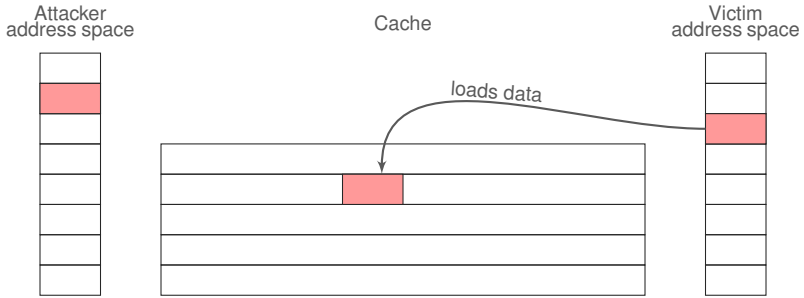
Flush+Reload



step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

Flush+Reload

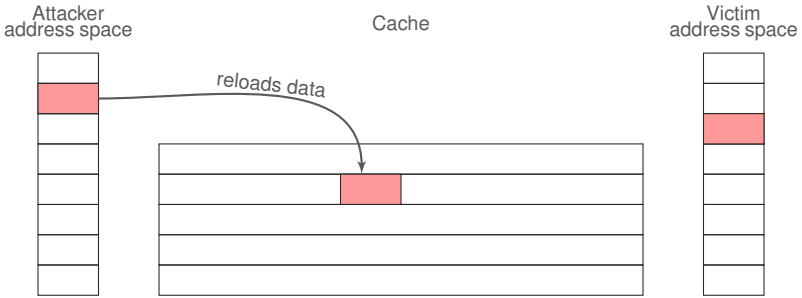


step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

Flush+Reload



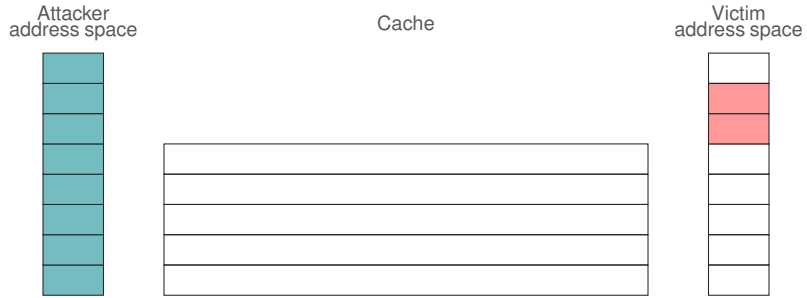
step 0: attacker maps shared library → shared memory, shared in cache

step 1: attacker flushes the shared line

step 2: victim loads data while performing encryption

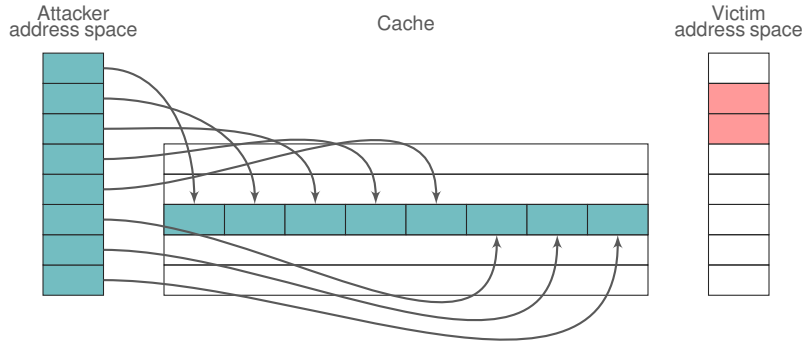
step 3: attacker reloads data → fast access if the victim loaded the line

Prime+Probe



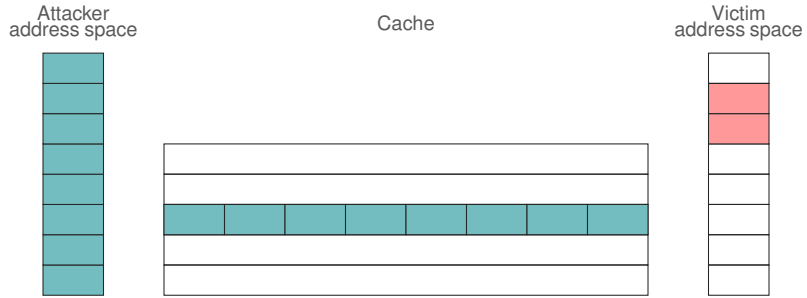
step 0: attacker fills the cache (prime)

Prime+Probe



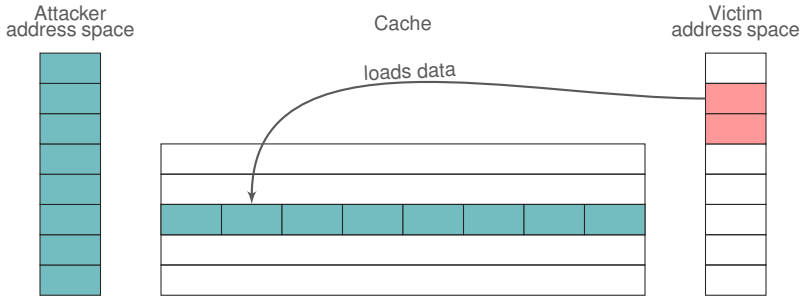
step 0: attacker fills the cache (prime)

Prime+Probe



step 0: attacker fills the cache (prime)

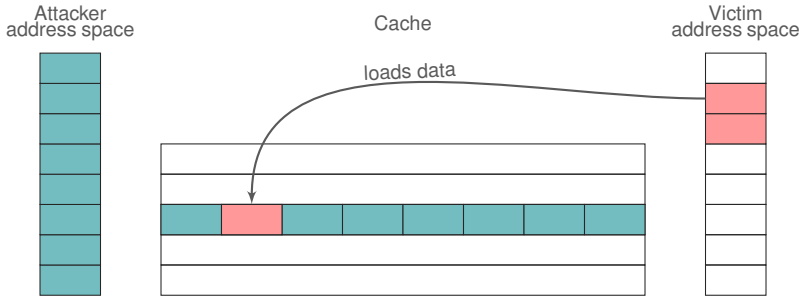
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

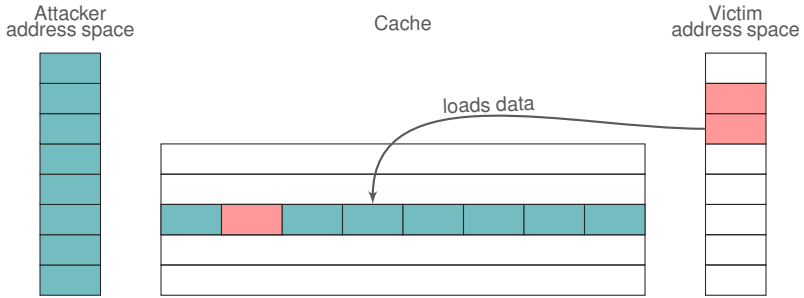
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

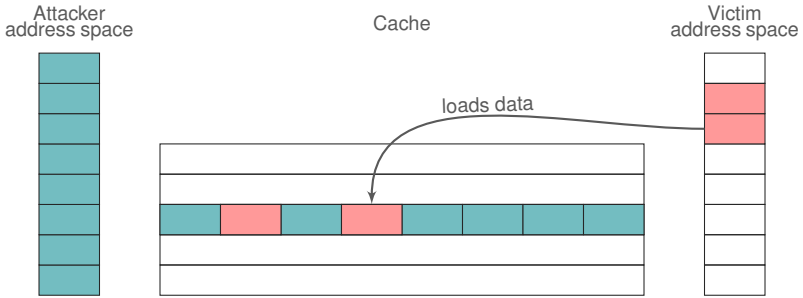
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

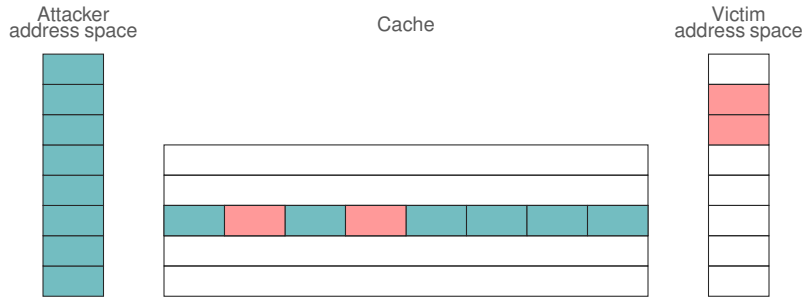
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

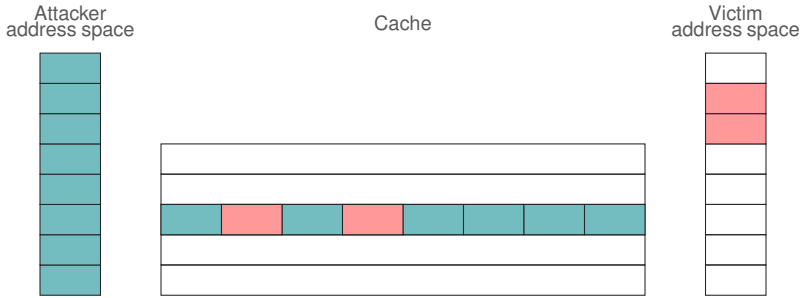
Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

Prime+Probe

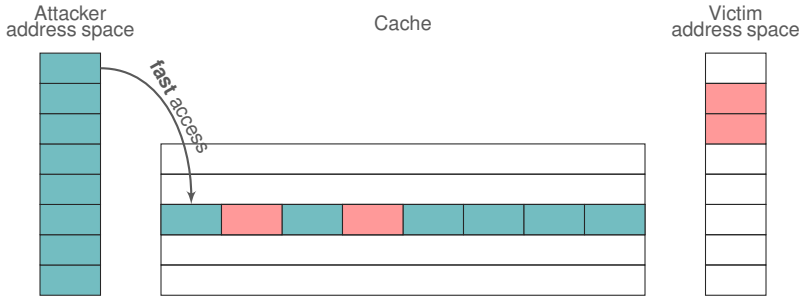


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe

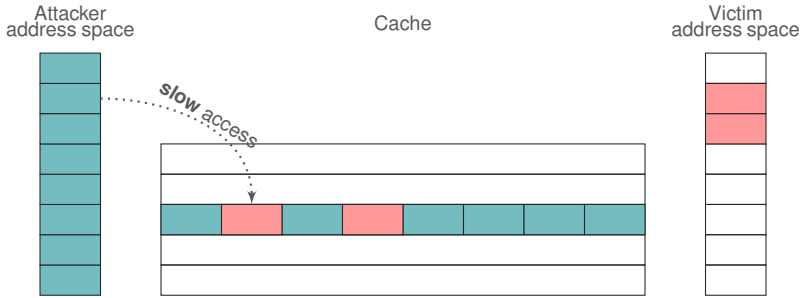


step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

Prime+Probe



step 0: attacker fills the cache (prime)

step 1: victim evicts cache lines while performing encryption

step 2: attacker probes data to determine if the set was accessed

1. Caches
2. Cache attacks
3. Cache Template Attacks
4. Rowhammer

Can we spy on gedit?

gedit source code tells us `gedit_window_get_active_view();` is used when a key is pressed, etc.

Can we spy on gedit?

`objdump -S /usr/bin/gedit`

Can we spy on gedit?

`objdump -S /usr/bin/gedit`

```
0000000000449ab0 <gedit_window_get_active_view>:
  449ab0: 53              : push   %rbx
  449ab1: 48 89 fb        : mov    %rdi,%rbx
  449ab4: e8 87 df ff ff  : callq  447a40 <gedit_window_get_type>
  449ab9: 48 85 db        : test   %rbx,%rbx
  449abc: 74 1c           : je     449ada <gedit_window_get_active_vie
  449abe: 48 8b 13        : mov    (%rbx),%rdx
  449ac1: 48 85 d2        : test   %rdx,%rdx
  449ac4: 74 05           : je     449acb <gedit_window_get_active_vie
```

Can we spy on gedit?

`objdump -S /usr/bin/gedit`

```
0000000000449ab0 <gedit_window_get_active_view>:
  449ab0: 53              : push   %rbx
  449ab1: 48 89 fb        : mov    %rdi,%rbx
  449ab4: e8 87 df ff ff  : callq  447a40 <gedit_window_get_type>
  449ab9: 48 85 db        : test   %rbx,%rbx
  449abc: 74 1c           : je     449ada <gedit_window_get_active_vie
  449abe: 48 8b 13        : mov    (%rbx),%rdx
  449ac1: 48 85 d2        : test   %rdx,%rdx
  449ac4: 74 05           : je     449acb <gedit_window_get_active_vie
```

Round `449ab0` up to be cache-line-aligned: `449ac0`

Where are my functions!?

`449ac0` is a virtual address! What now?

Where in the binary is this function?

Where are my functions!?

```
readelf -a /usr/bin/gedit
```

Where are my functions!?

```
readelf -a /usr/bin/gedit

Program Headers:
Type  Offset    VirtAddr  PhysAddr
      FileSiz   MemSiz     Flags  Align
...
LOAD  0x000000  0x400000  0x400000
      0x089314  0x089314   R E    200000
...
```

Where are my functions!?

```
readelf -a /usr/bin/gedit

Program Headers:
Type  Offset    VirtAddr  PhysAddr
      FileSiz   MemSiz     Flags  Align
...
LOAD  0x000000  0x400000  0x400000
      0x089314  0x089314   R E    200000
...
```

`0x400000-0x489314` is `0x0-0x89314` in the binary

Where are my functions!?

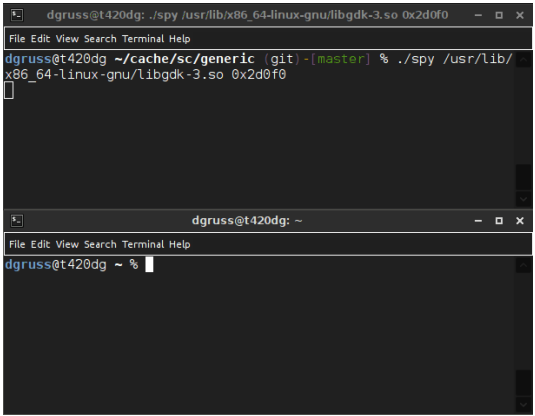
```
readelf -a /usr/bin/gedit

Program Headers:
Type  Offset    VirtAddr  PhysAddr
      FileSiz   MemSiz     Flags  Align
...
LOAD  0x000000  0x400000  0x400000
      0x089314  0x089314   R E    200000
...
```

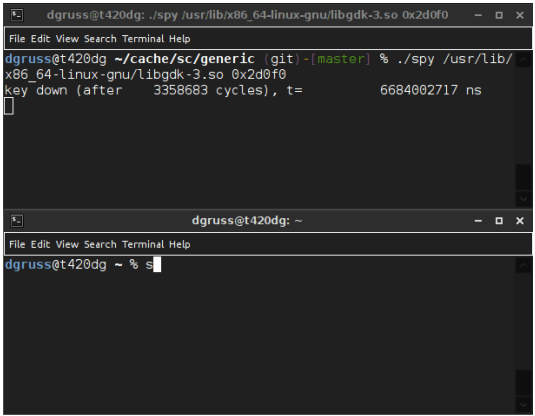
`0x400000-0x489314` is `0x0-0x89314` in the binary

$\rightarrow$ we attack `0x49ac0`

A cache-based keystroke logger?

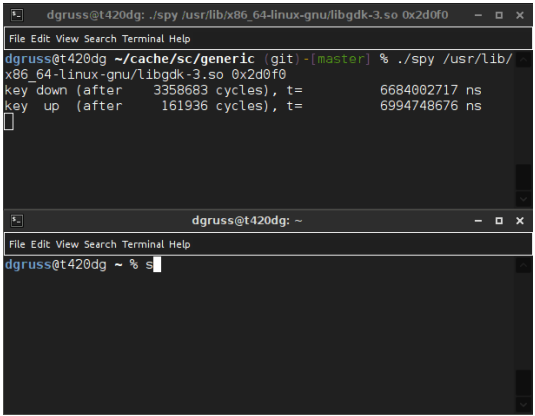


A cache-based keystroke logger?

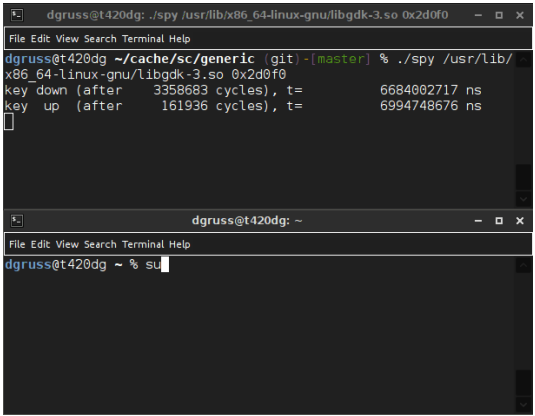


Daniel Gruss, IAIK, Graz University of Technology

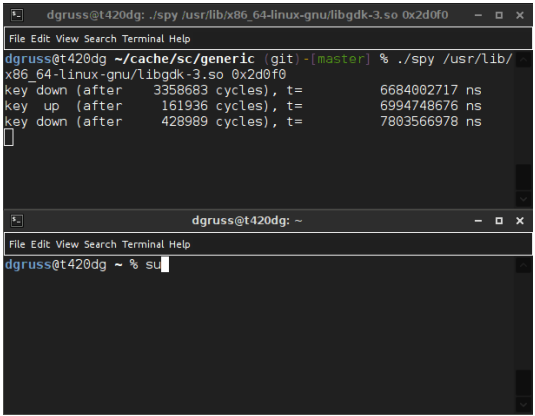
A cache-based keystroke logger?



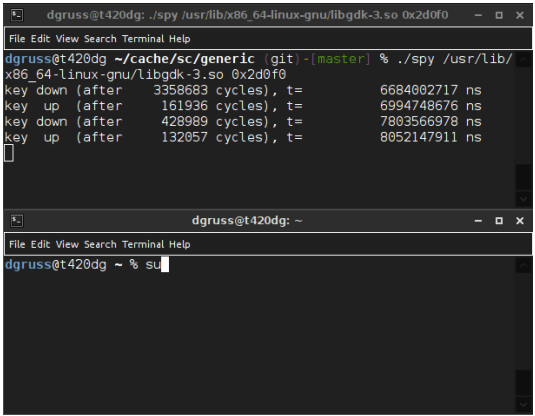
A cache-based keystroke logger?



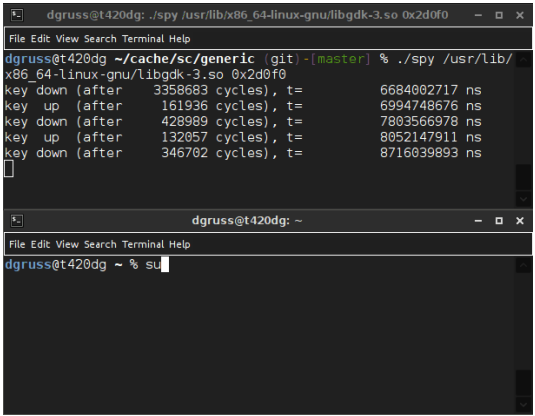
A cache-based keystroke logger?



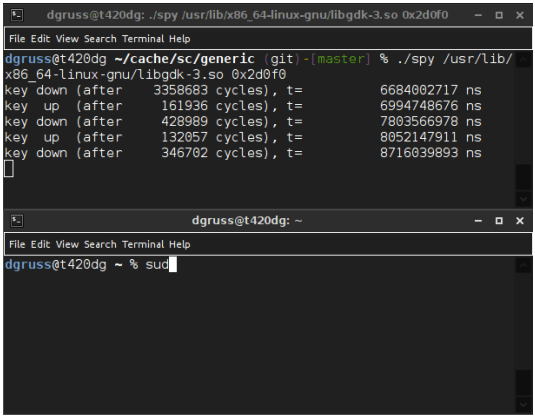
A cache-based keystroke logger?



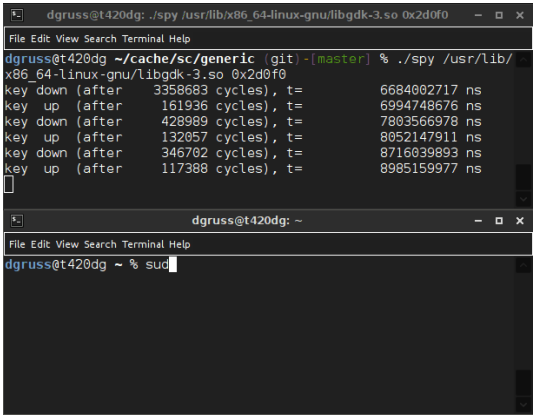
A cache-based keystroke logger?



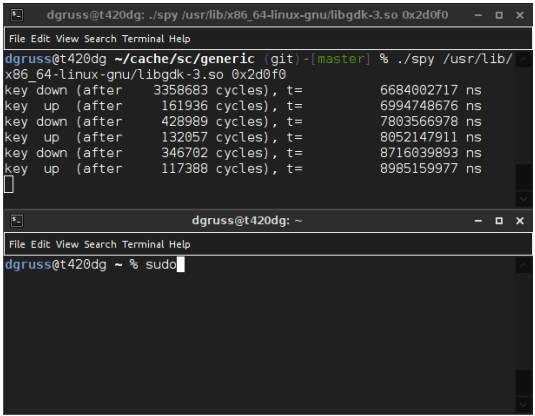
A cache-based keystroke logger?



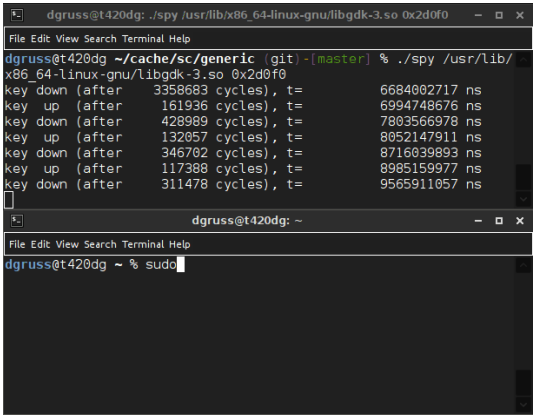
A cache-based keystroke logger?



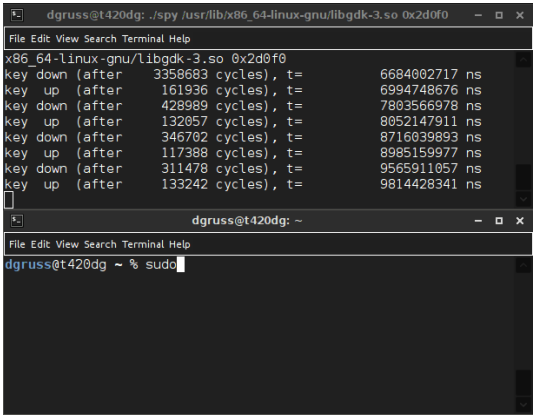
A cache-based keystroke logger?



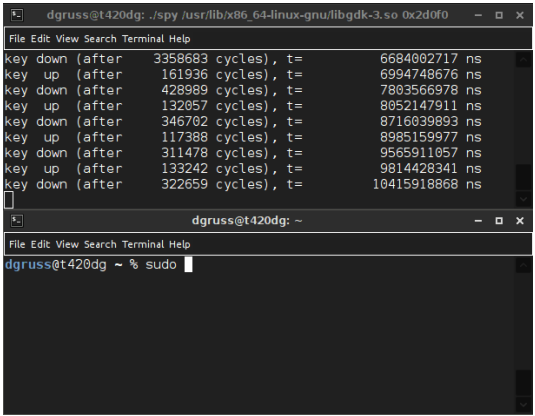
A cache-based keystroke logger?



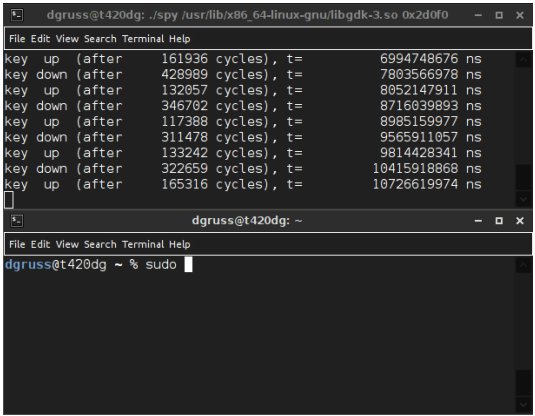
A cache-based keystroke logger?



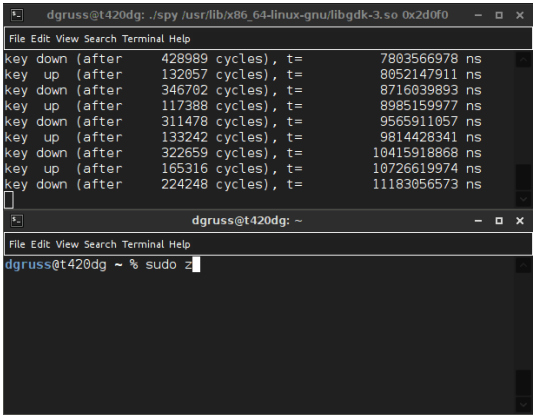
A cache-based keystroke logger?



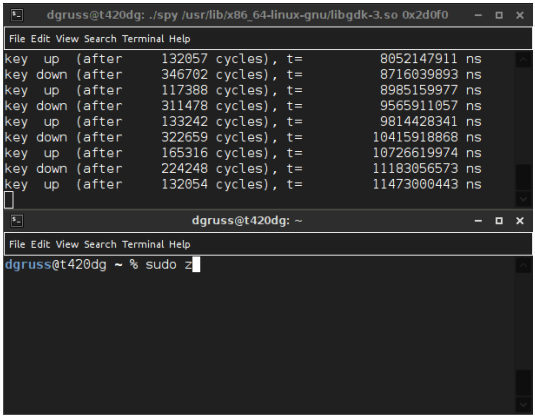
A cache-based keystroke logger?



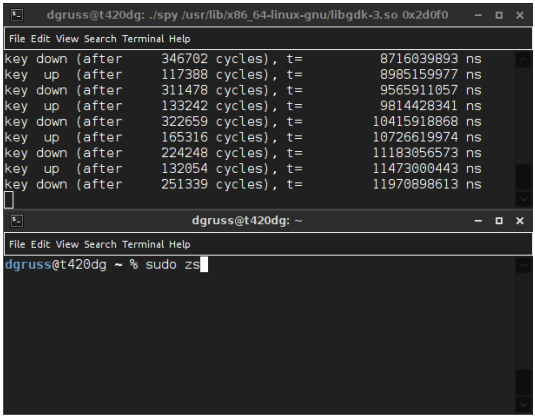
A cache-based keystroke logger?



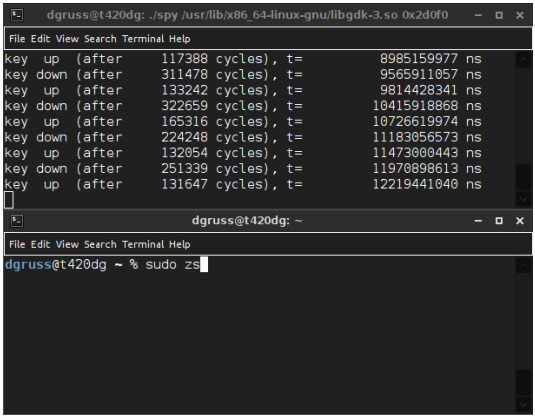
A cache-based keystroke logger?



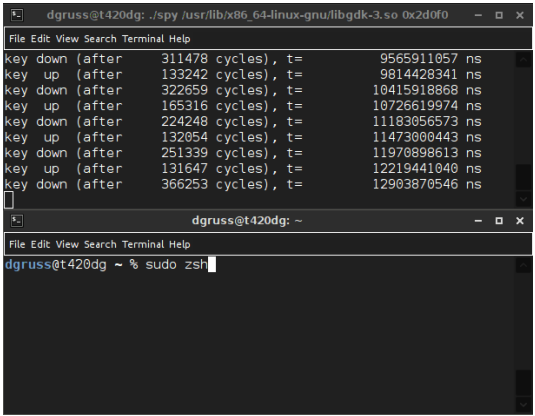
A cache-based keystroke logger?



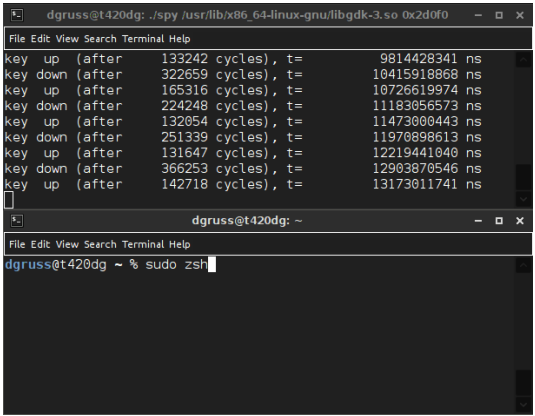
A cache-based keystroke logger?



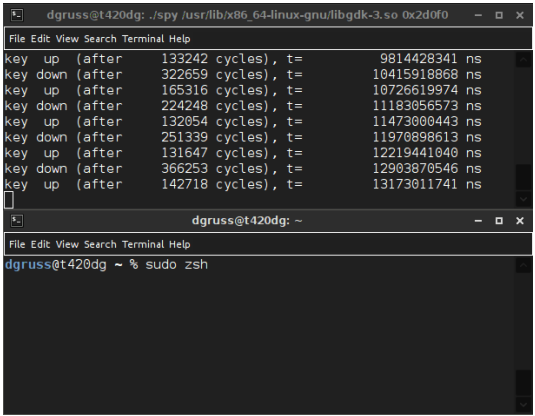
A cache-based keystroke logger?



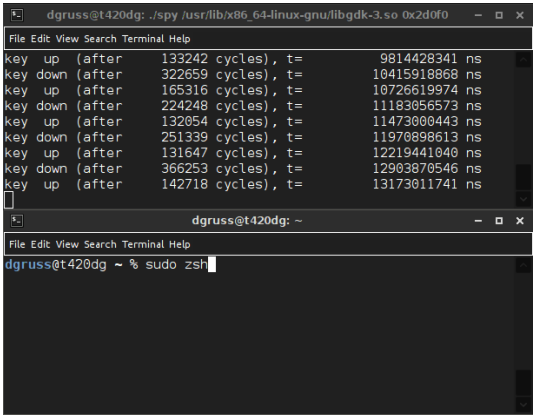
A cache-based keystroke logger?



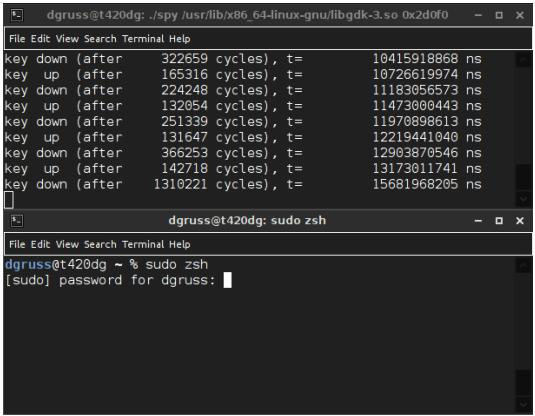
A cache-based keystroke logger?



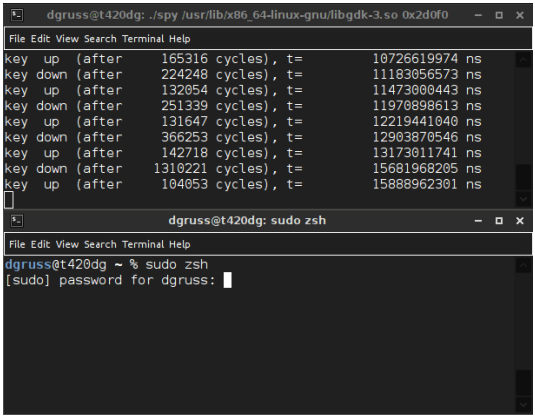
A cache-based keystroke logger?



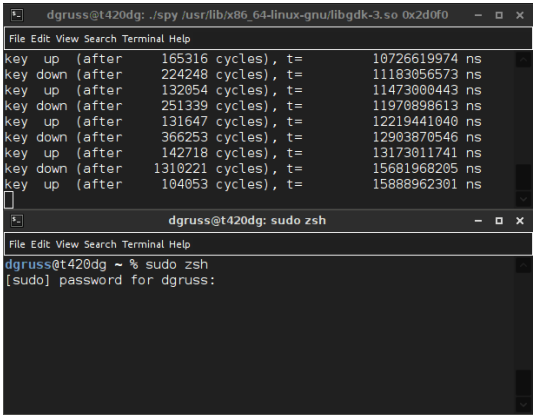
A cache-based keystroke logger?



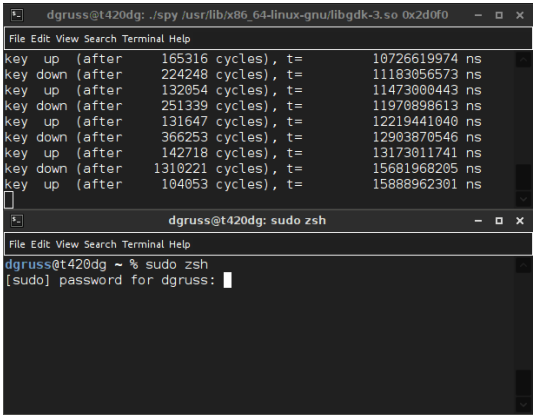
A cache-based keystroke logger?



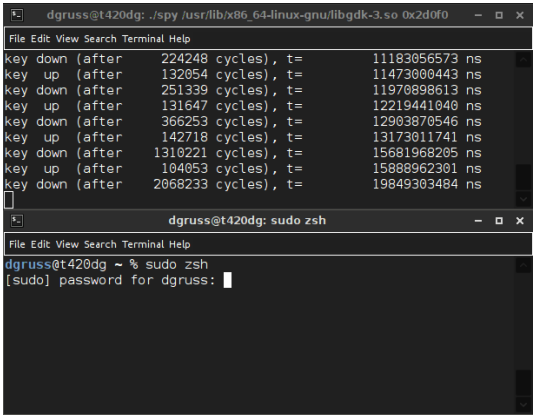
A cache-based keystroke logger?



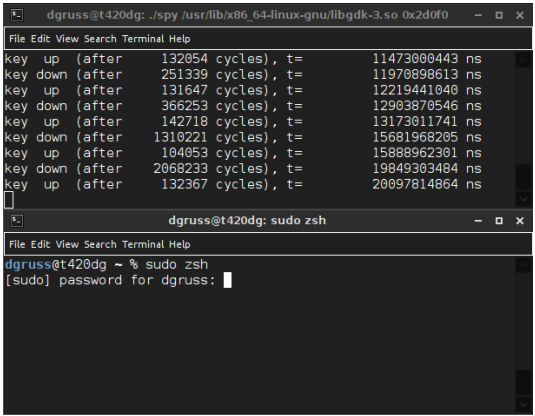
A cache-based keystroke logger?



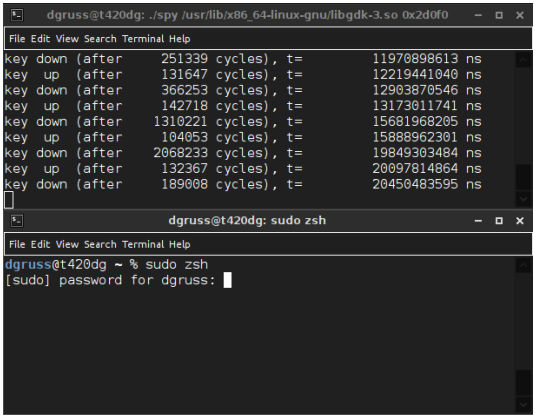
A cache-based keystroke logger?



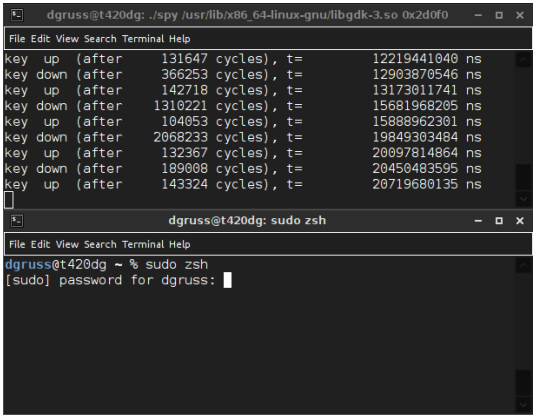
A cache-based keystroke logger?



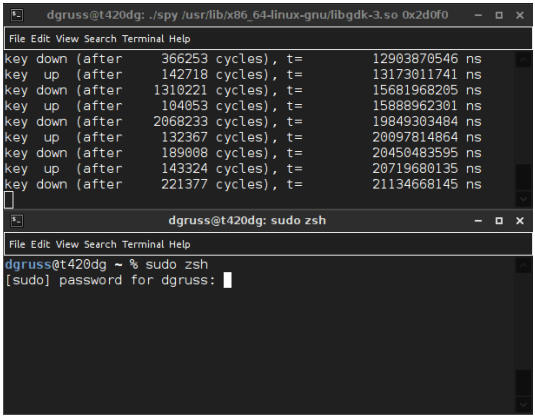
A cache-based keystroke logger?



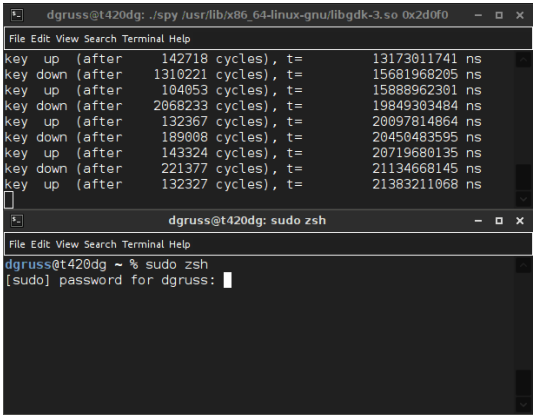
A cache-based keystroke logger?



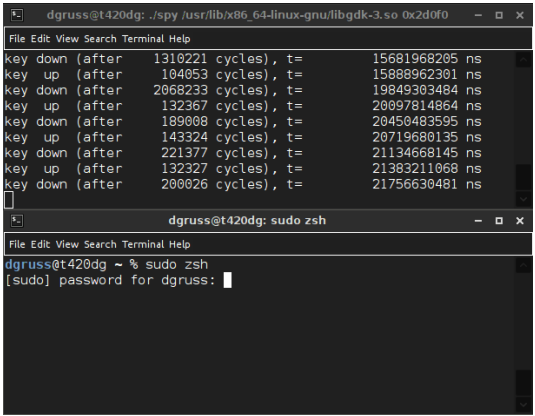
A cache-based keystroke logger?



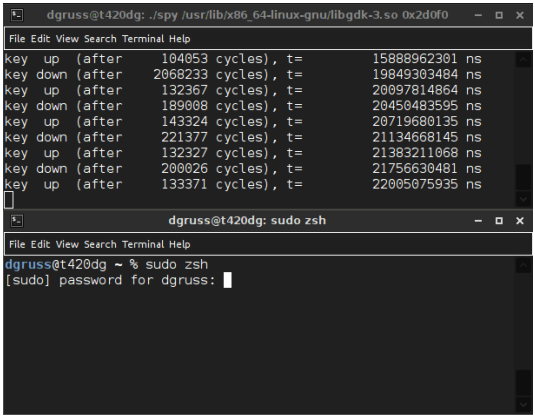
A cache-based keystroke logger?



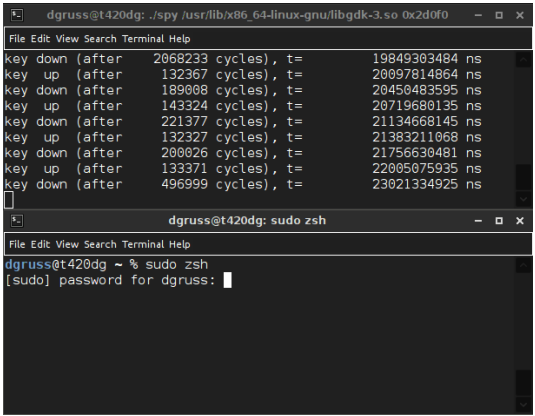
A cache-based keystroke logger?



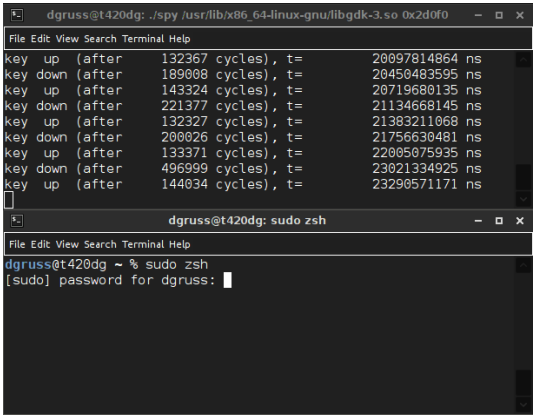
A cache-based keystroke logger?



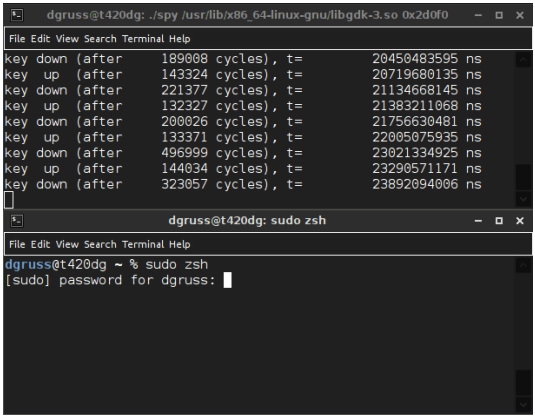
A cache-based keystroke logger?



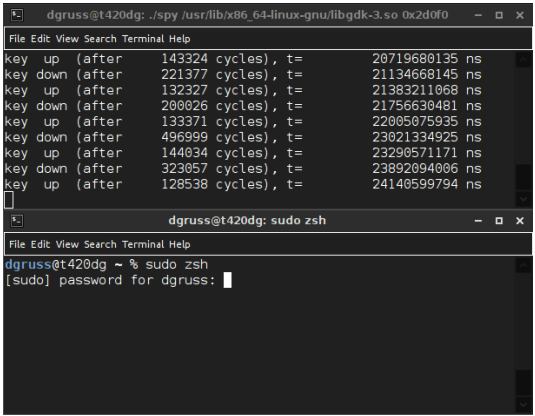
A cache-based keystroke logger?



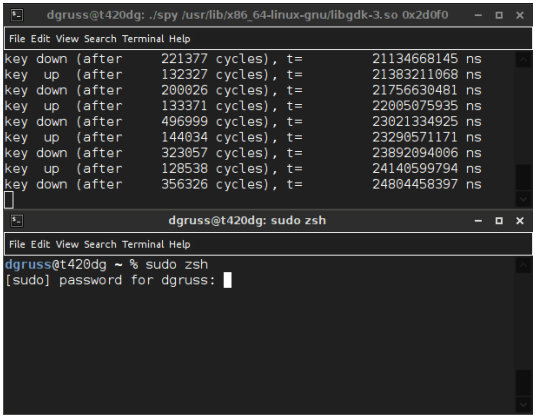
A cache-based keystroke logger?



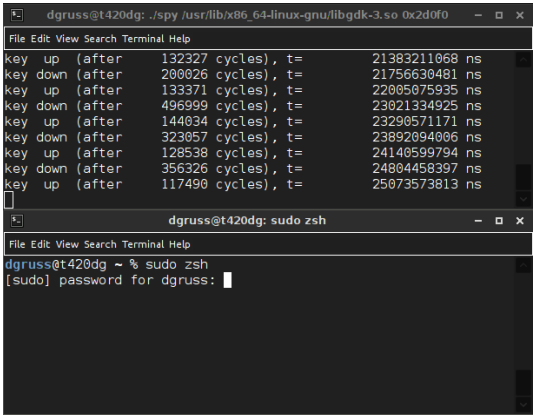
A cache-based keystroke logger?



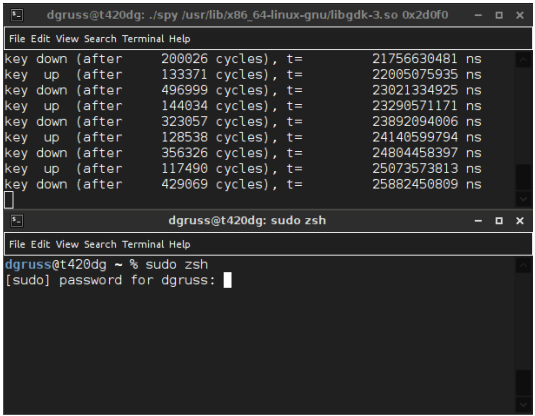
A cache-based keystroke logger?



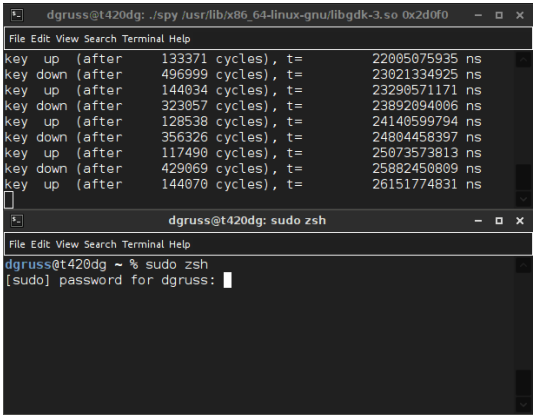
A cache-based keystroke logger?



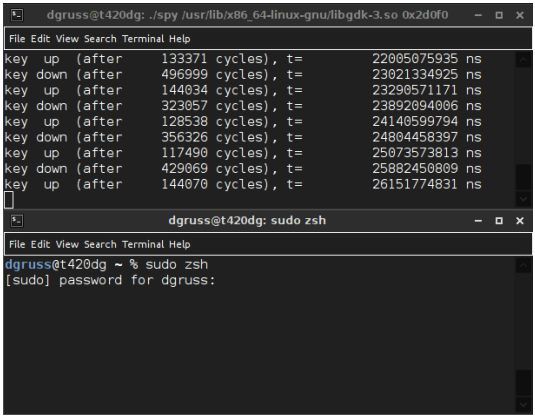
A cache-based keystroke logger?



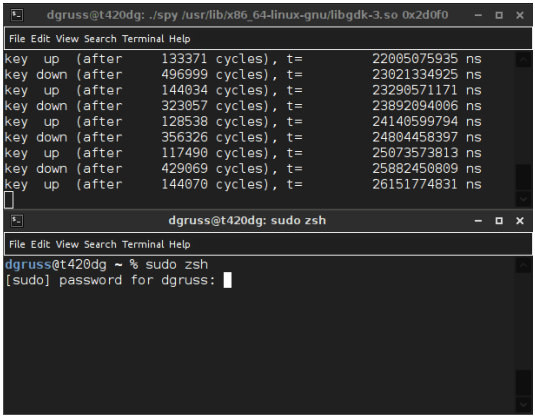
A cache-based keystroke logger?



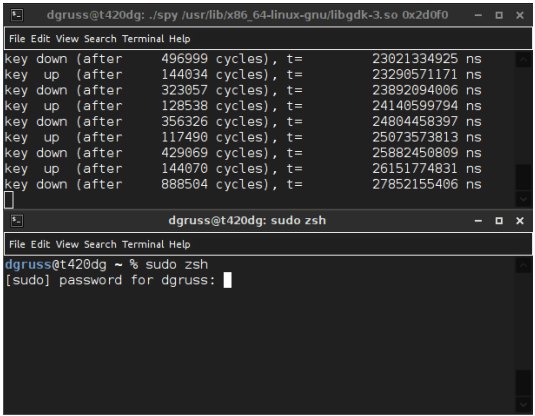
A cache-based keystroke logger?



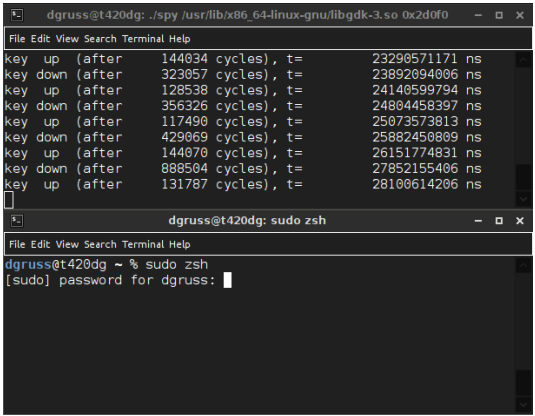
A cache-based keystroke logger?



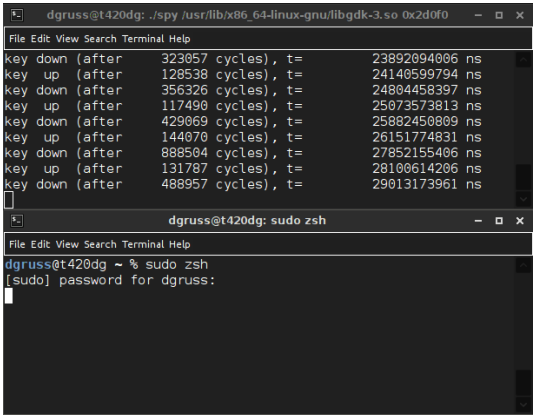
A cache-based keystroke logger?



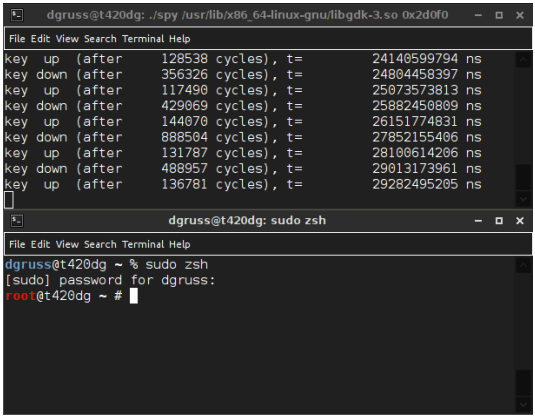
A cache-based keystroke logger?



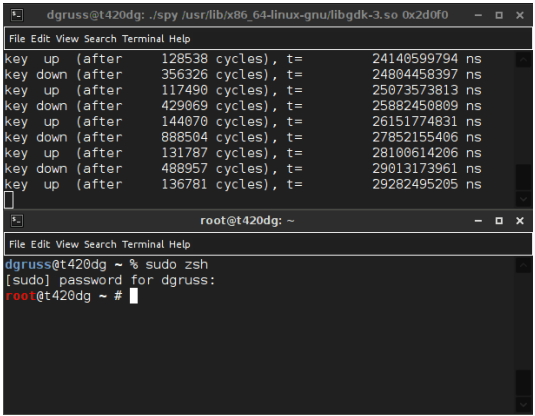
A cache-based keystroke logger?



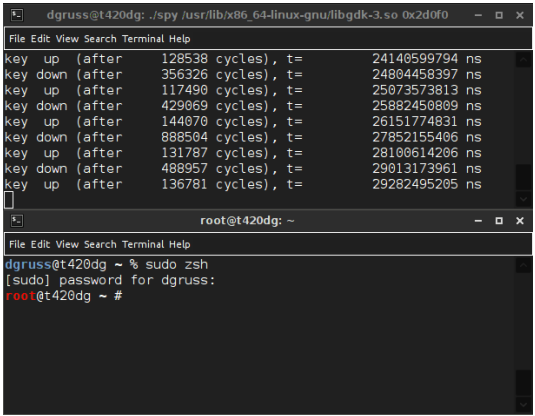
A cache-based keystroke logger?



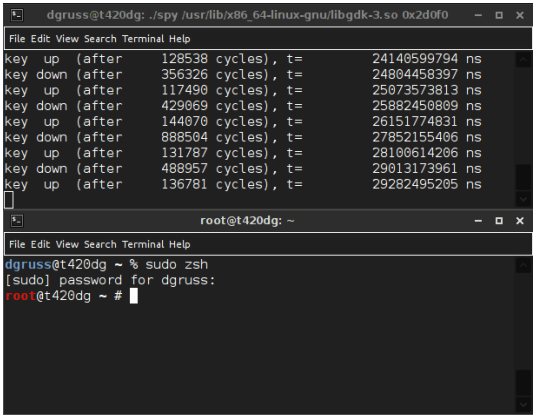
A cache-based keystroke logger?



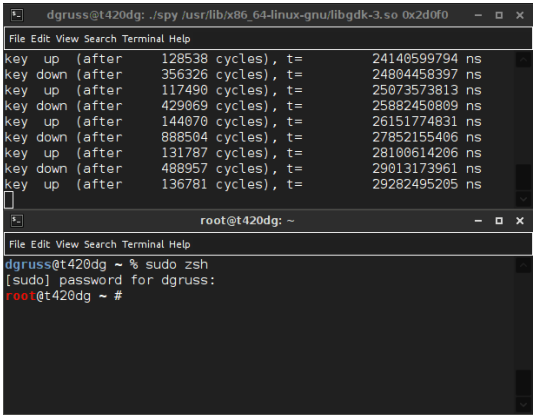
A cache-based keystroke logger?



A cache-based keystroke logger?



A cache-based keystroke logger?



Challenges

- How to locate key-dependent memory accesses?

Challenges

- How to locate key-dependent memory accesses?
- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries

Challenges

- How to locate key-dependent memory accesses?
- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries
- Difficult to find all exploitable addresses

Lessons learnt

- Staring at objdump's for weeks will give you headache

Lessons learnt

- Staring at objdump's for weeks will give you headache
- Automate everything right from the start

Cache Template Attacks

- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software

Cache Template Attacks

- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software
- Examples:
  - Cache-based keylogger
  - Automatic attacks on crypto algorithms

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Cache Template Attacks

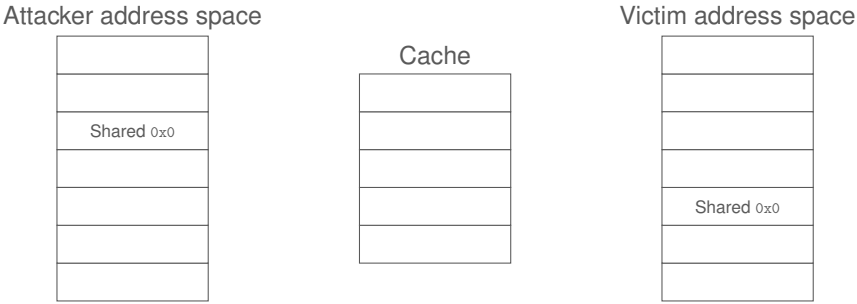
Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

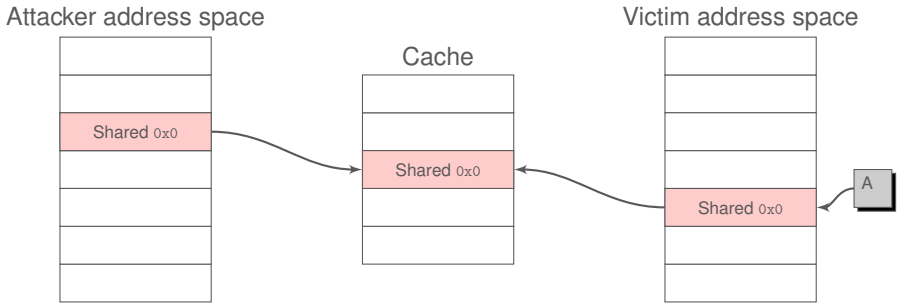
- Monitor exploitable addresses

Profiling Phase



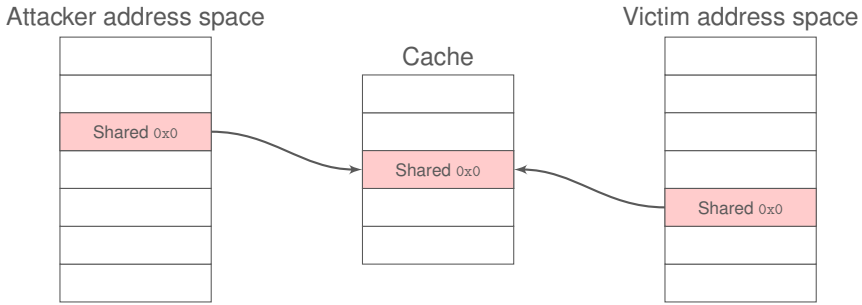
Cache is empty

Profiling Phase



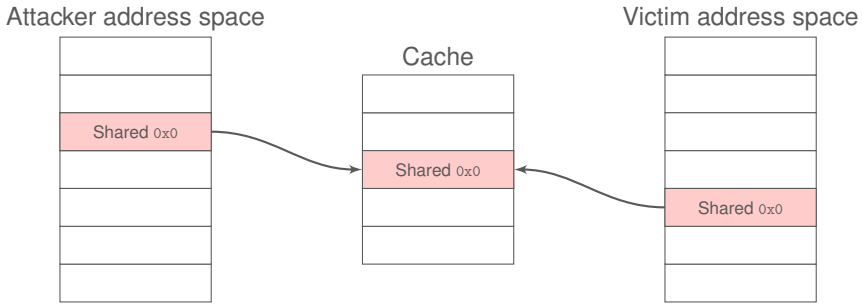
Attacker triggers an event

Profiling Phase



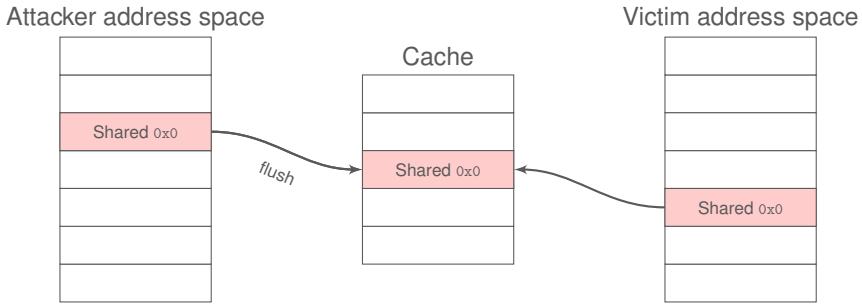
Attacker checks one address for cache hits (“Reload”)

Profiling Phase



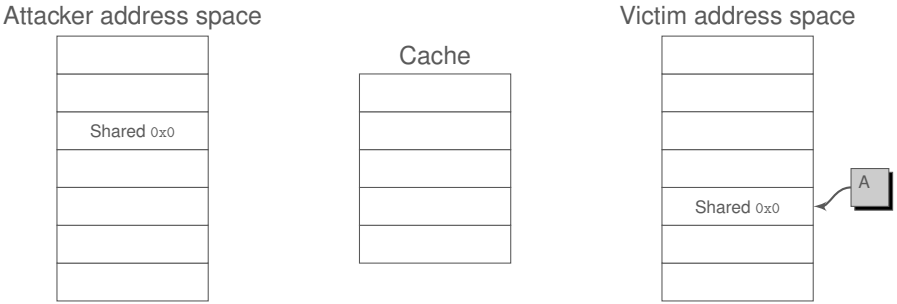
Update cache hit ratio (per event and address)

Profiling Phase



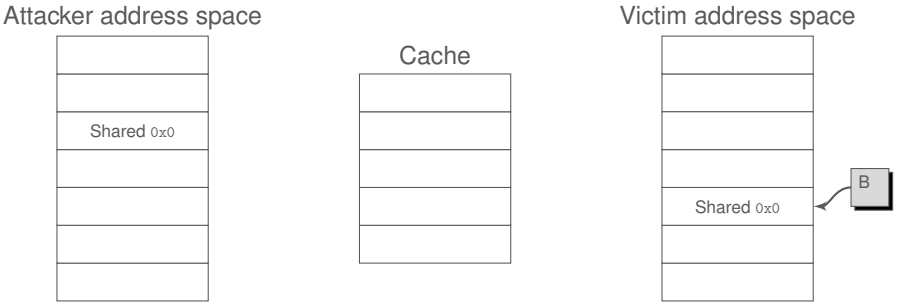
Attacker flushes shared memory

Profiling Phase



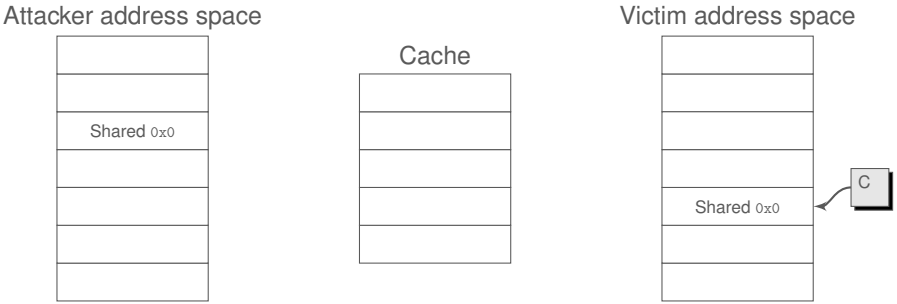
Repeat for higher accuracy

Profiling Phase



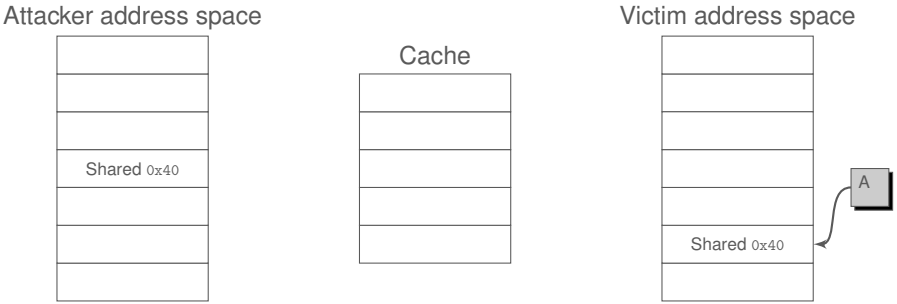
Repeat for all events

Profiling Phase



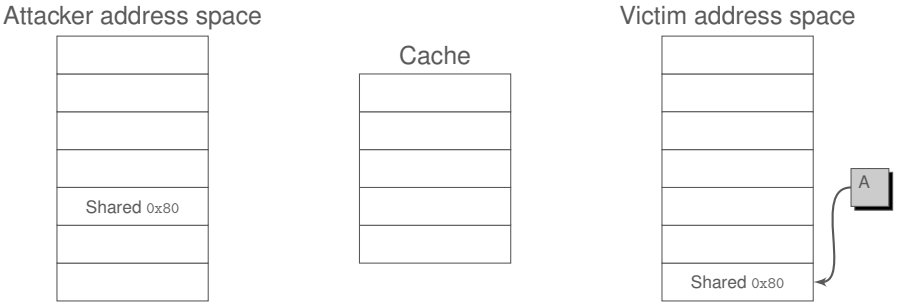
Repeat for all events

Profiling Phase



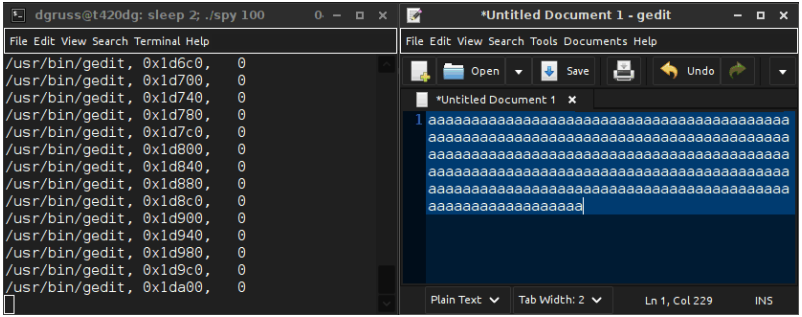
Continue with next address

Profiling Phase

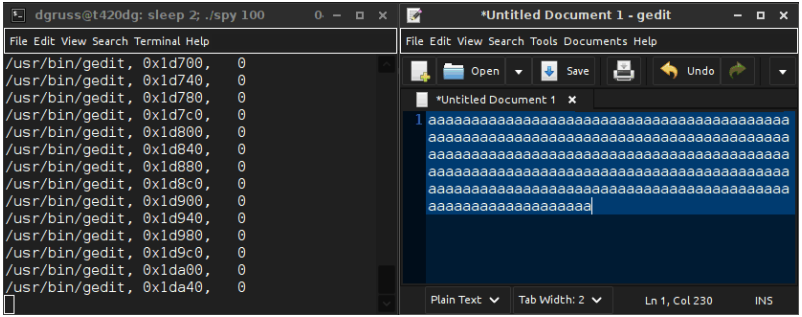


Continue with next address

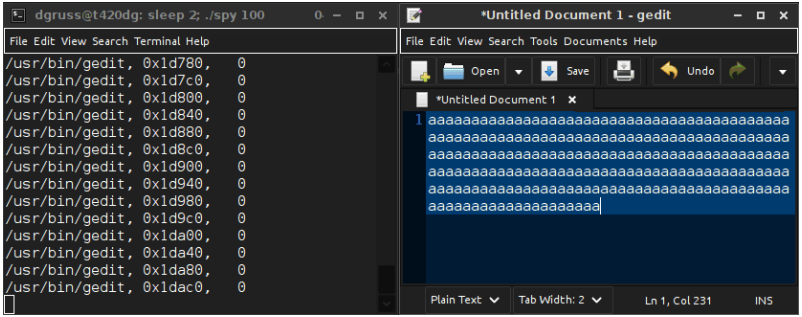
Profiling a Single Event



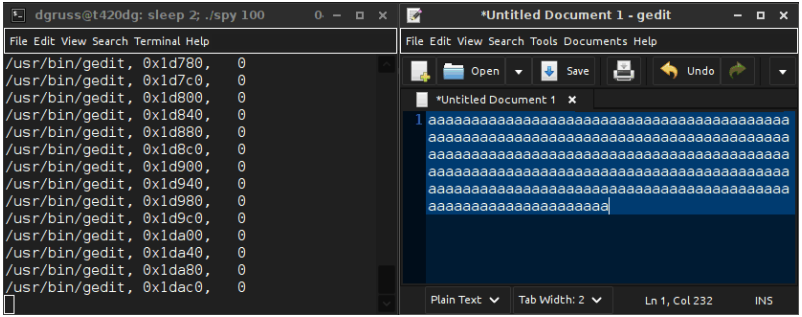
Profiling a Single Event



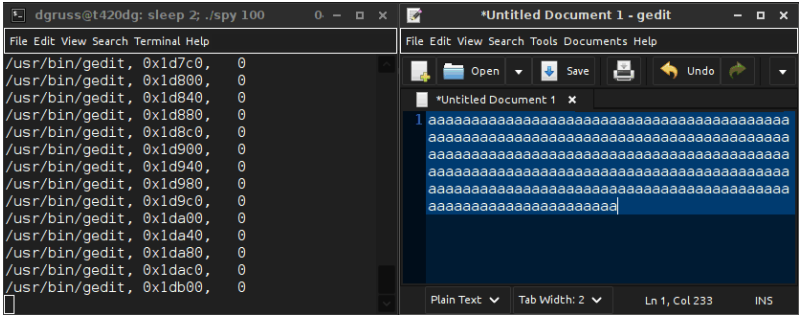
Profiling a Single Event



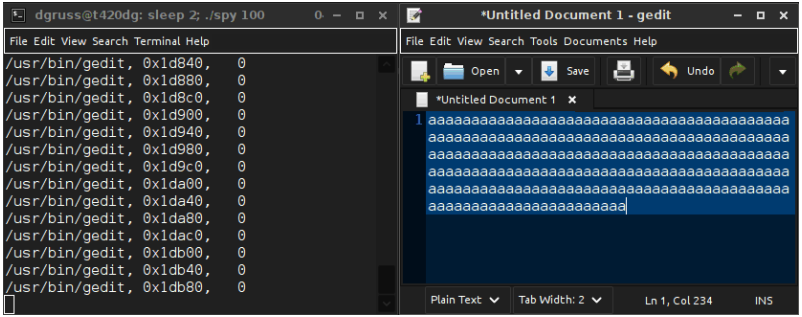
Profiling a Single Event



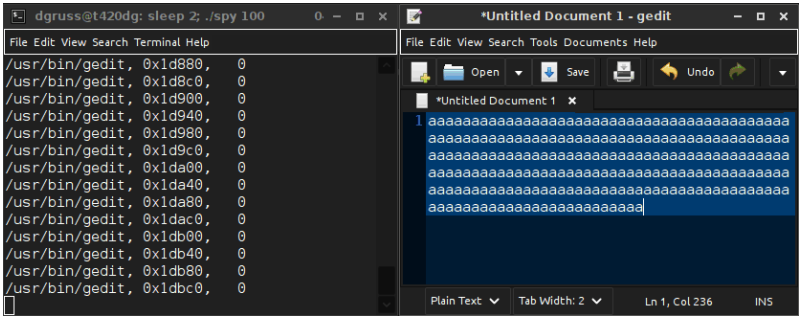
Profiling a Single Event



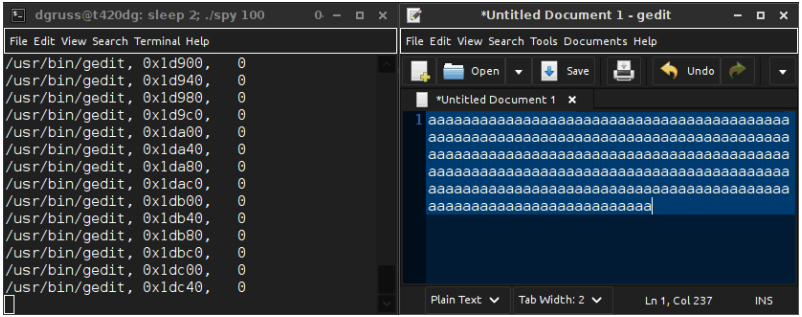
Profiling a Single Event



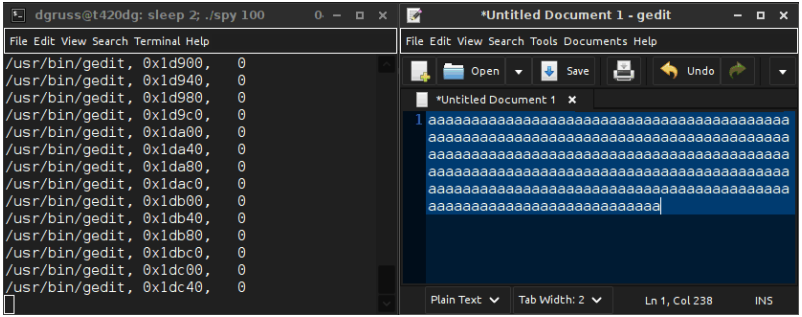
Profiling a Single Event



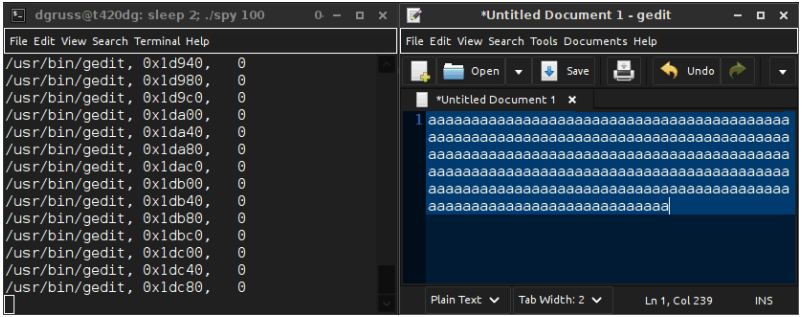
Profiling a Single Event



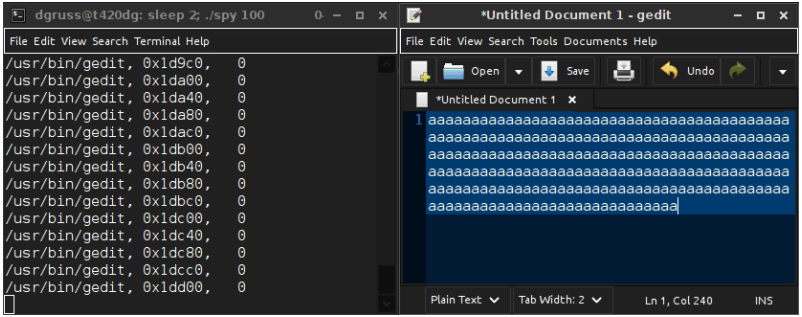
Profiling a Single Event



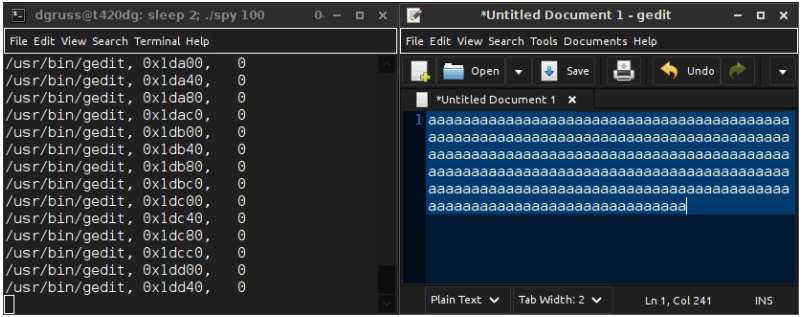
Profiling a Single Event



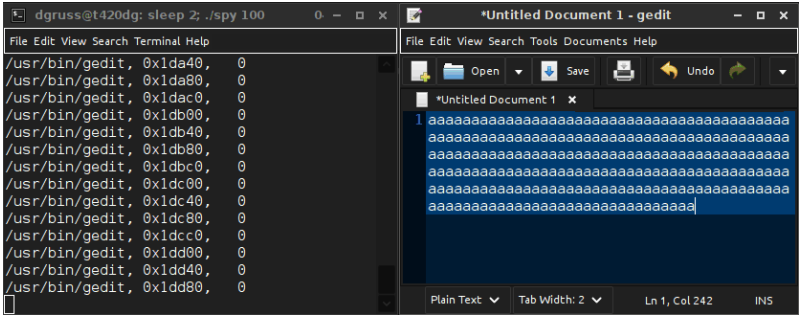
Profiling a Single Event



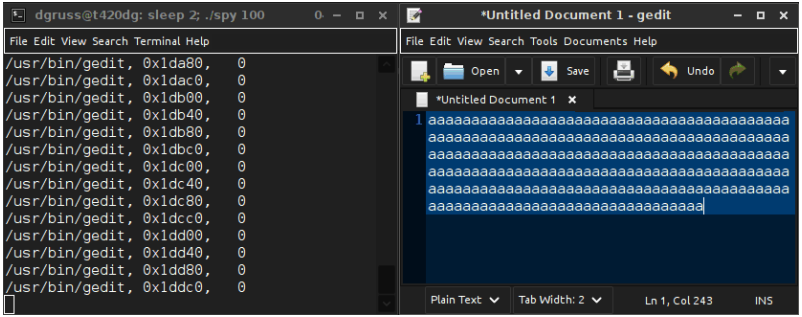
Profiling a Single Event



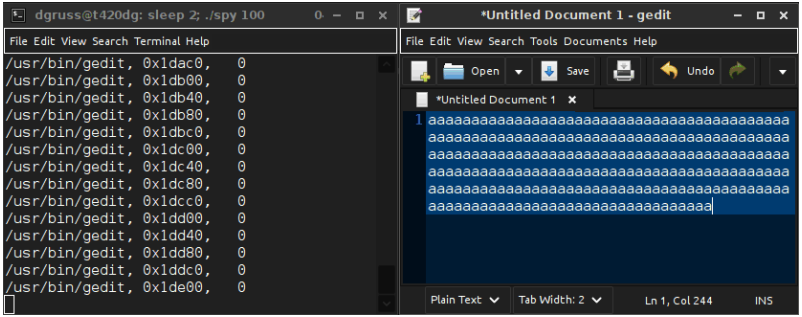
Profiling a Single Event



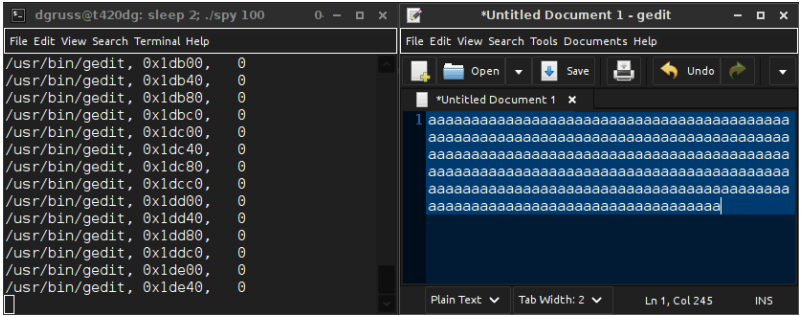
Profiling a Single Event



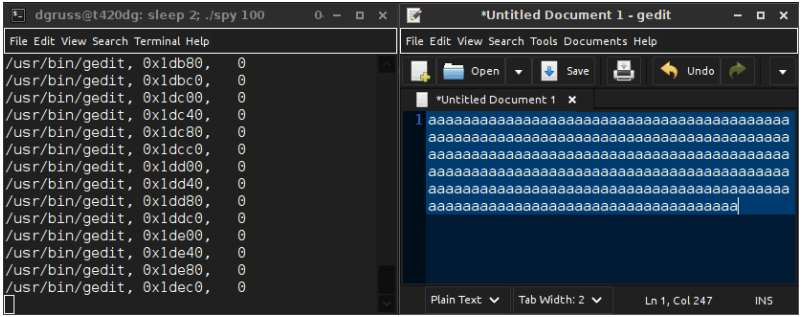
Profiling a Single Event



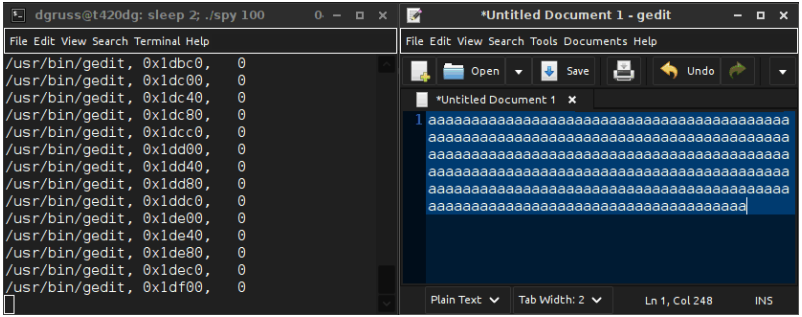
Profiling a Single Event



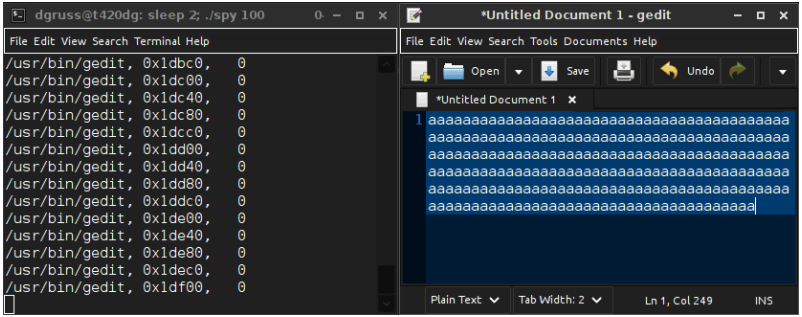
Profiling a Single Event



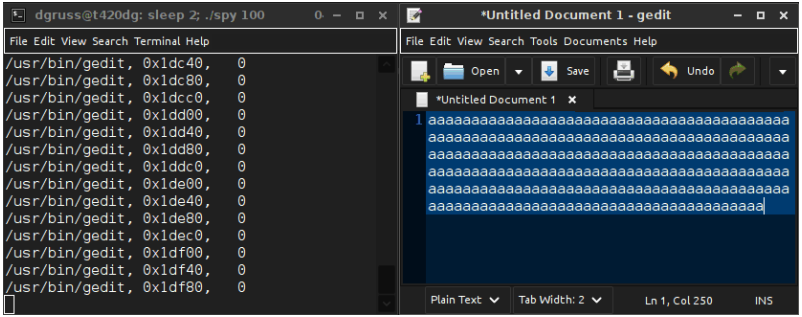
Profiling a Single Event



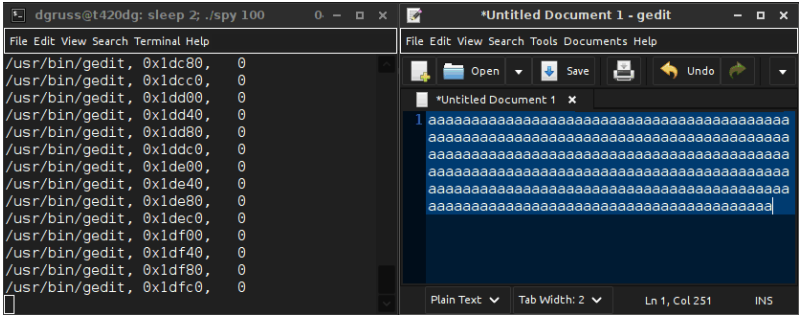
Profiling a Single Event



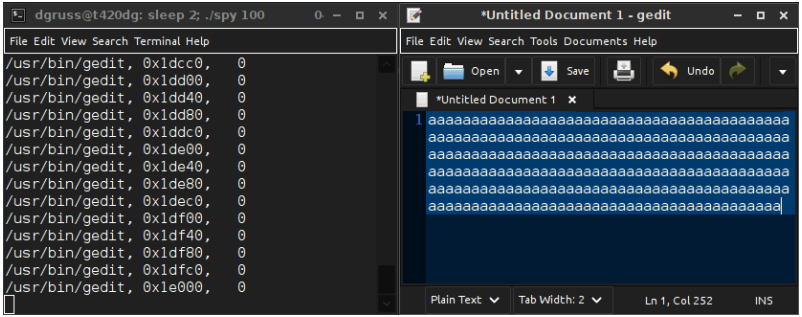
Profiling a Single Event



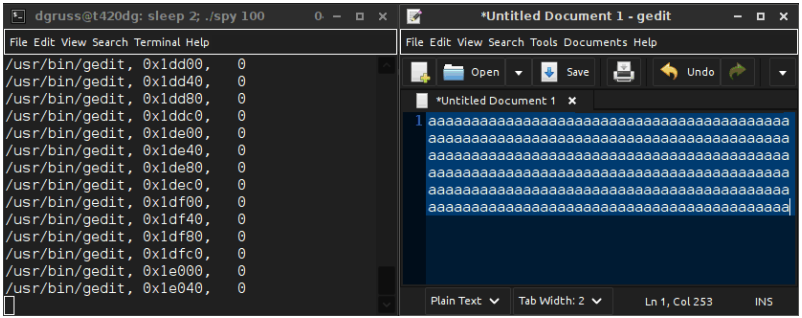
Profiling a Single Event



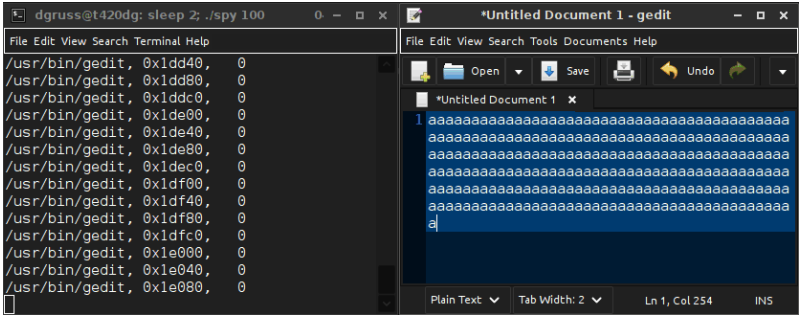
Profiling a Single Event



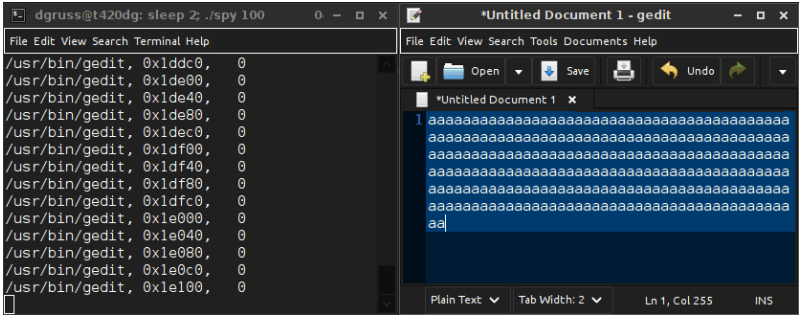
Profiling a Single Event



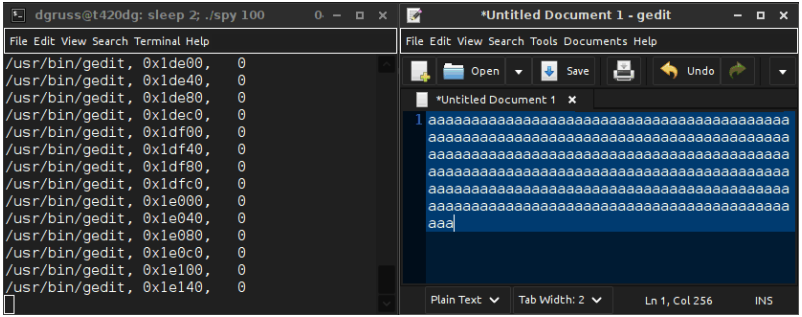
Profiling a Single Event



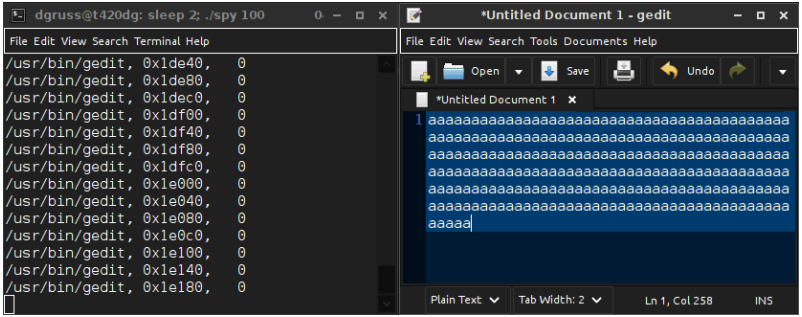
Profiling a Single Event



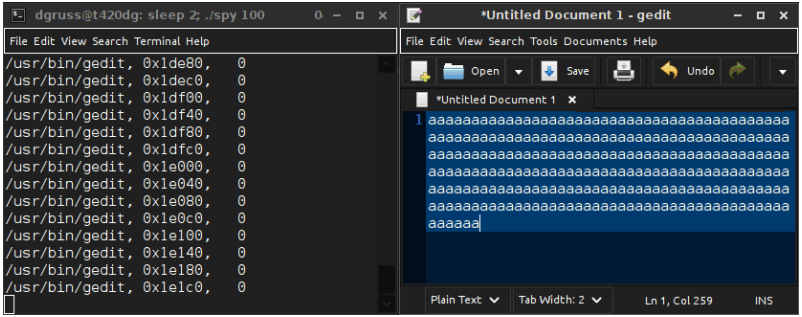
Profiling a Single Event



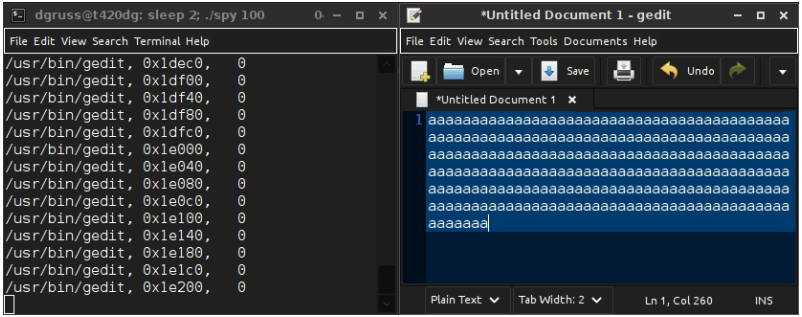
Profiling a Single Event



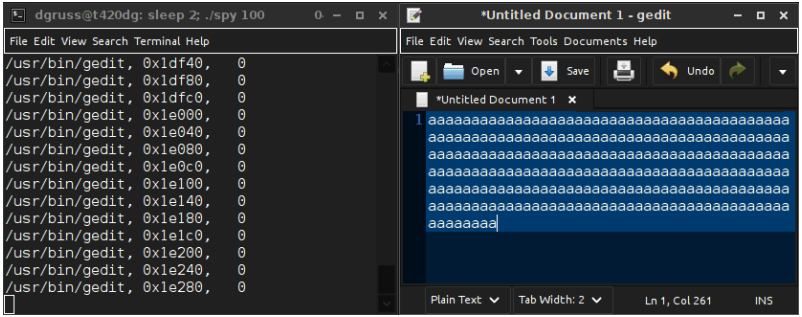
Profiling a Single Event



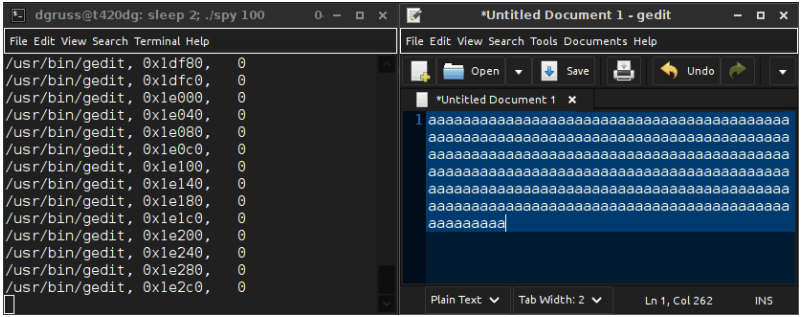
Profiling a Single Event



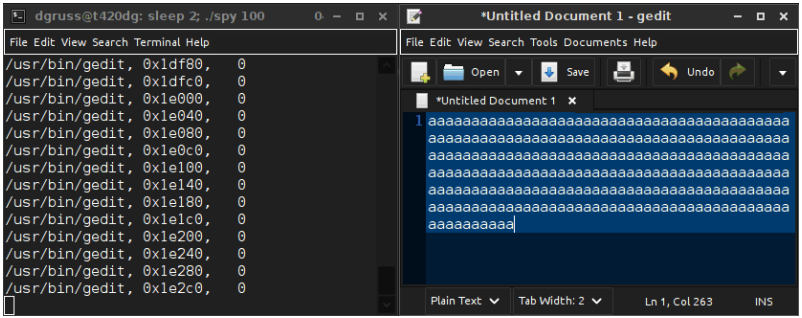
Profiling a Single Event



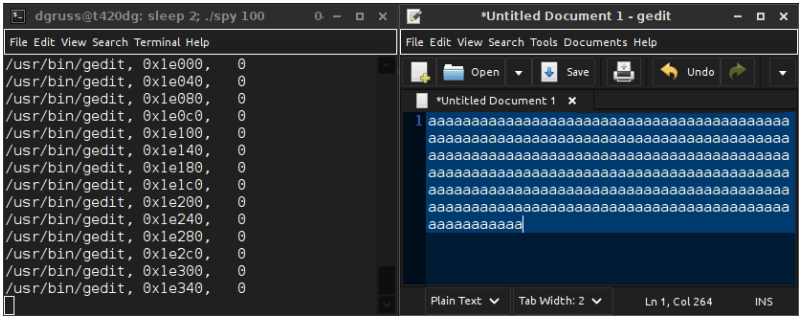
Profiling a Single Event



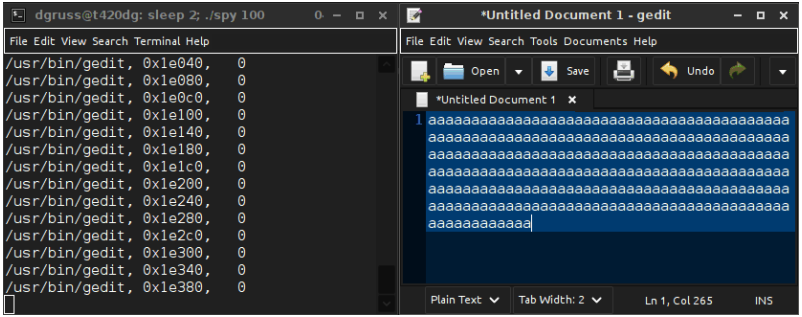
Profiling a Single Event



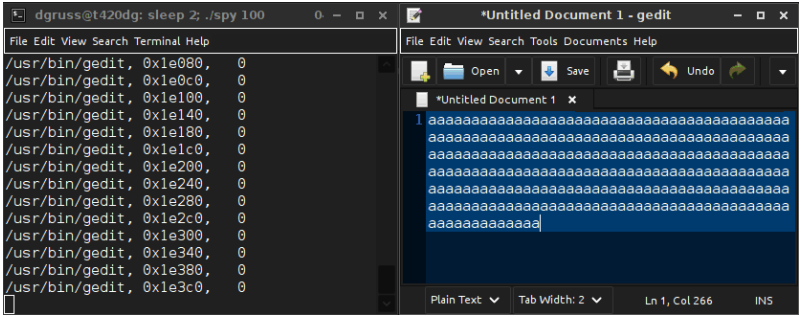
Profiling a Single Event



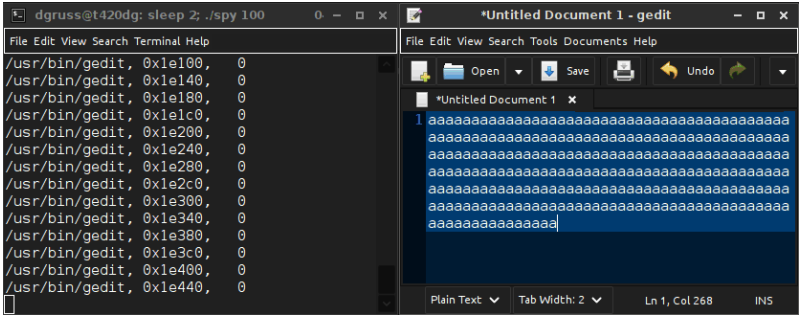
Profiling a Single Event



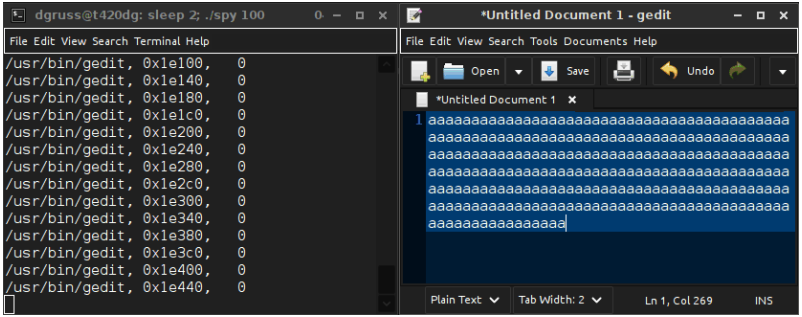
Profiling a Single Event



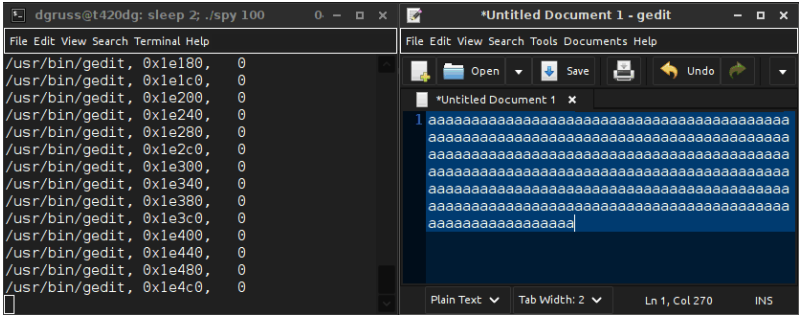
Profiling a Single Event



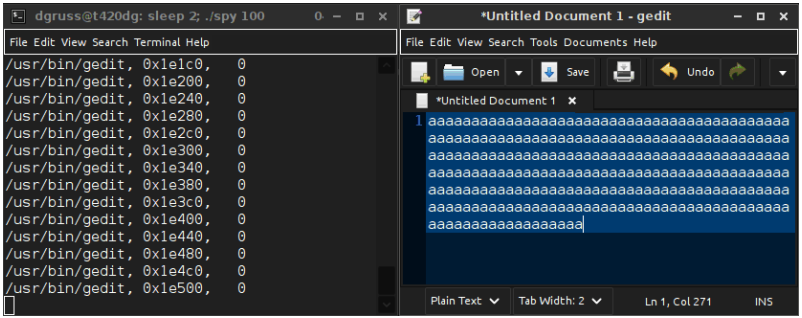
Profiling a Single Event



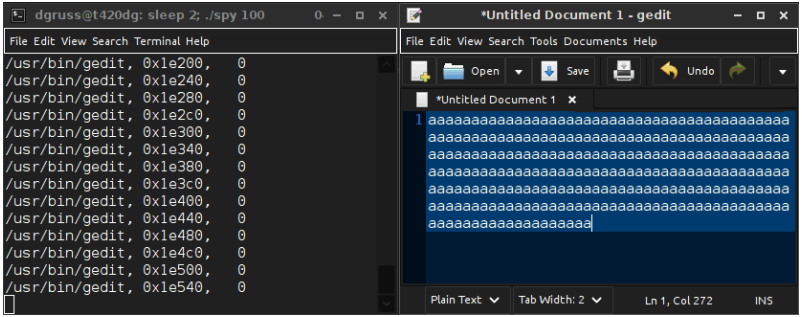
Profiling a Single Event



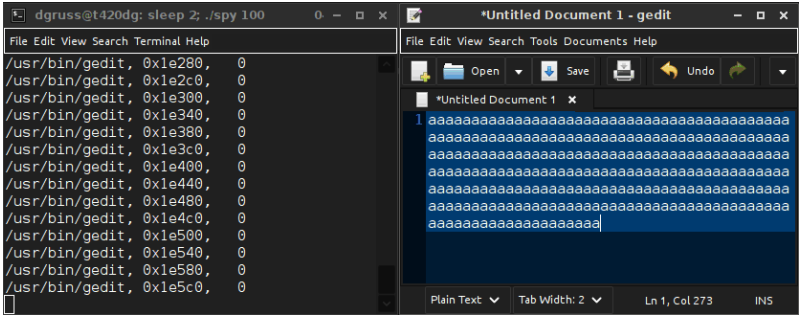
Profiling a Single Event



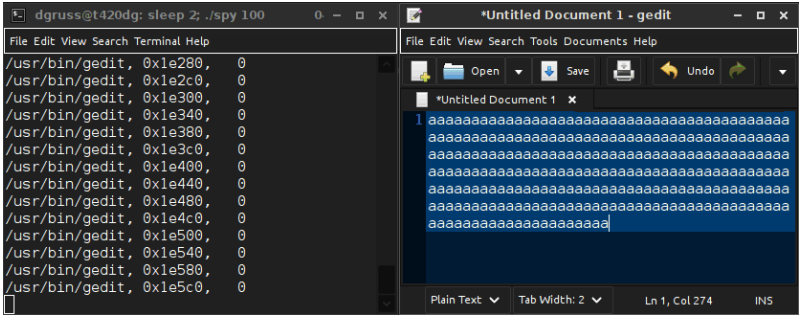
Profiling a Single Event



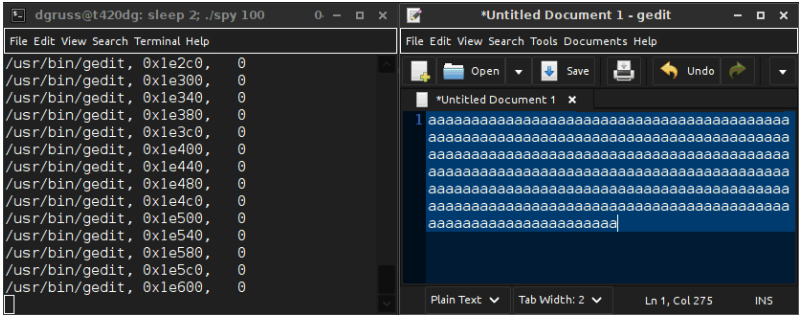
Profiling a Single Event



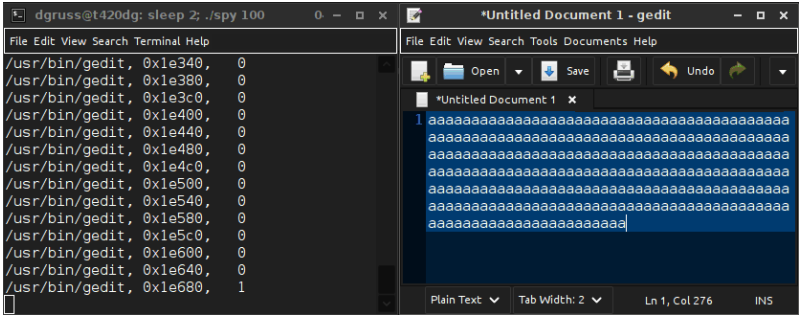
Profiling a Single Event



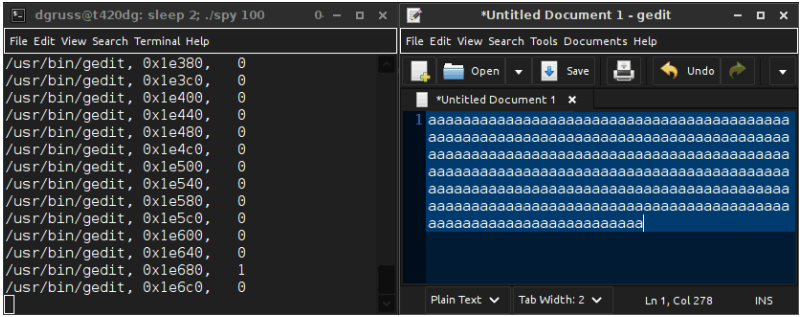
Profiling a Single Event



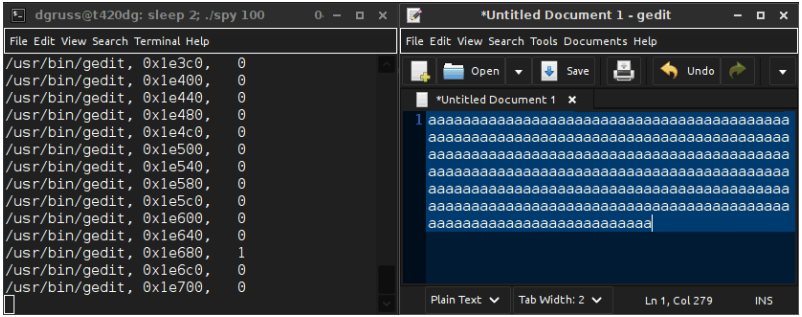
Profiling a Single Event



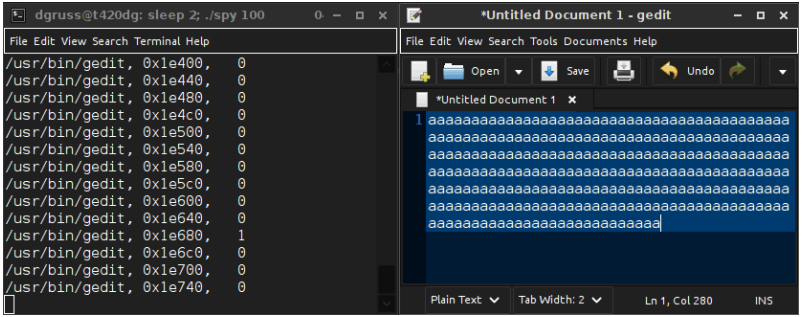
Profiling a Single Event



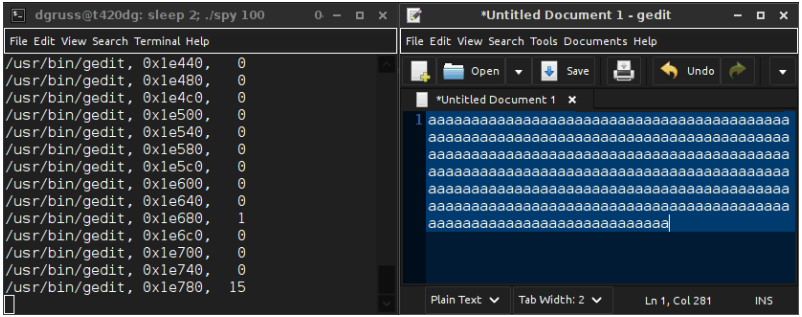
Profiling a Single Event



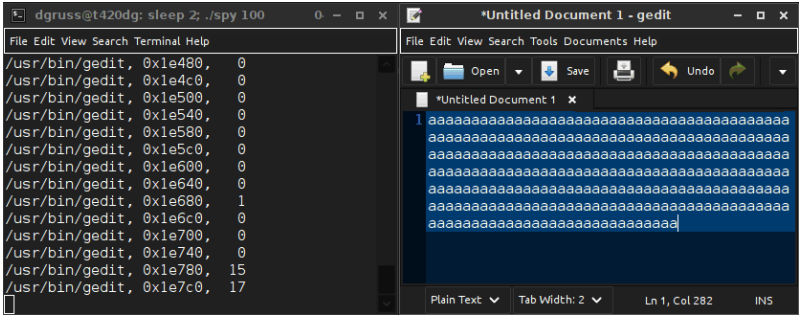
Profiling a Single Event



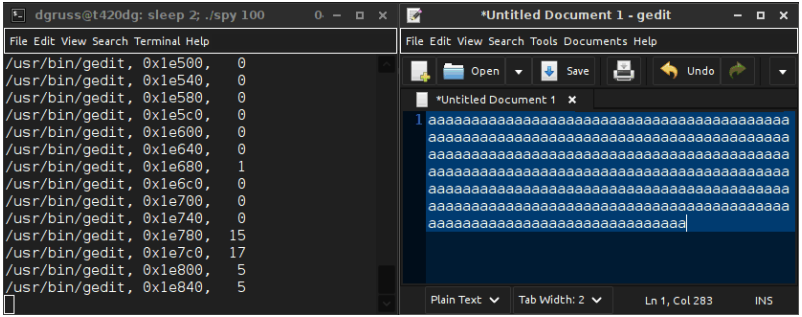
Profiling a Single Event



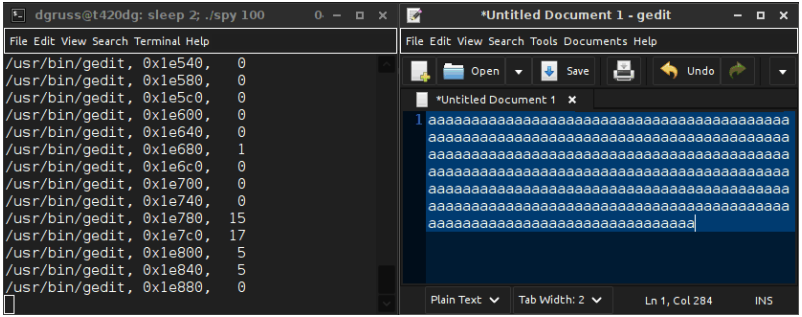
Profiling a Single Event



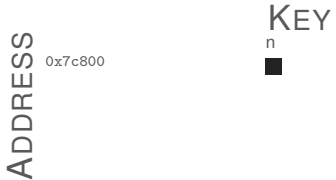
Profiling a Single Event



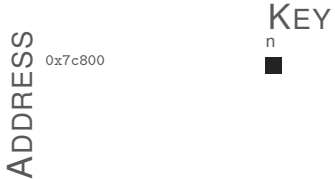
Profiling a Single Event



Profiling Phase: 1 Event, 1 Address



Profiling Phase: 1 Event, 1 Address



Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{n})$: 200 / 200

Profiling Phase: All Events, 1 Address

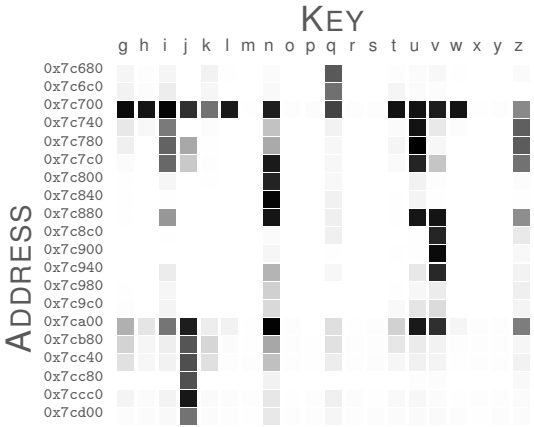
Profiling Phase: All Events, 1 Address

Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{u})$: 13 / 200

Profiling Phase: All Events, 1 Address

Distinguish `n` from other keys by monitoring `0x7c800`

Profiling Phase: All Events, All Addresses



Daniel Gruss, IAIK, Graz University of Technology

Exploitation Phase

- Monitor addresses from Cache Template

Exploitation Phase

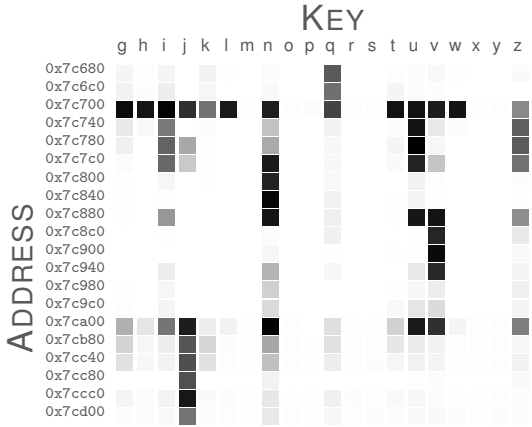
- Monitor addresses from Cache Template
- Report to log file / attacker

Exploitation Phase

- Monitor addresses from Cache Template
- Report to log file / attacker
- Manual analysis of log file
  - Find password in keypress log, etc.

Example Attack: Keylogging

- Linux with GTK: monitor keystrokes of specific keys
- Detect groups of keys
- Some keys distinct



Daniel Gruss, IAIK, Graz University of Technology

Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte

Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

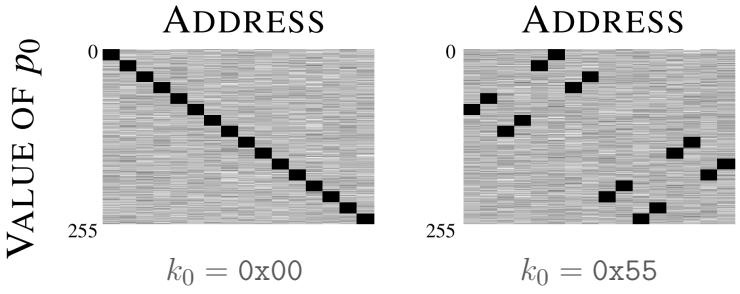
- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event

Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase: an auto-generated first-round attack

Example Attack: AES T-Tables Template



Cache Template Attacks: How hard is it?

- We let students attack different programs
- Leakage typically found within 30 minutes
- Wide range of programs

Cache Template Attacks: Example Applications

- Keystrokes in nano, quasselclient, xterm, Pidgin, gksu, Firefox, GNU Emacs (on OSX), mousepad, user32.dll on Windows
- Keystrokes specifically in the password field in KeePassX
- Font rendering in libharfbuzz (used by many Linux applications)
- Action buttons in VLC (play, mute, ...)
- Receiving characters in netcat
- Distinguishing keygroups in gksu
- Monitoring checkbox clicking in gksu

Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage

Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries

Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries
- Large scale automated cache attacks possible!

Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries
- Large scale automated cache attacks possible!
- But: still requires shared memory

What about Prime+Probe?

- Good: no shared memory

What about Prime+Probe?

- Good: no shared memory
- Bad: much more difficult due to noise

What about Prime+Probe?

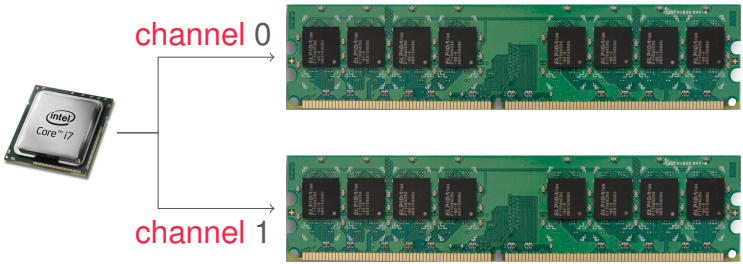
- Good: no shared memory
- Bad: much more difficult due to noise

→ Let's do something more fun with Prime+Probe!

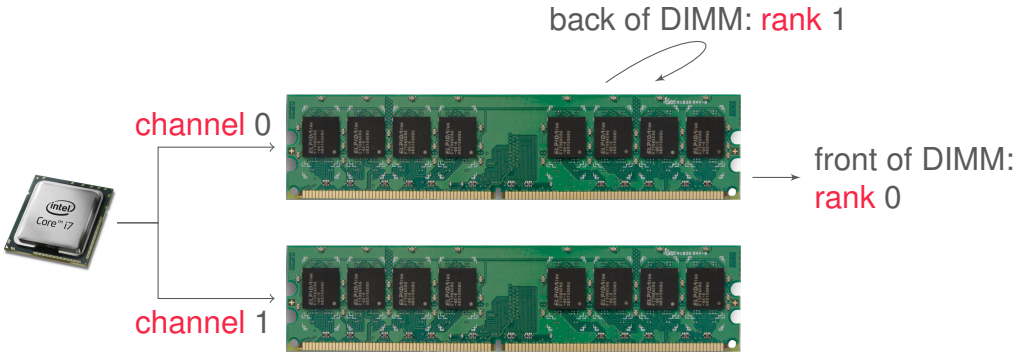
1. Caches
2. Cache attacks
3. Cache Template Attacks
4. Rowhammer

DRAM organization example

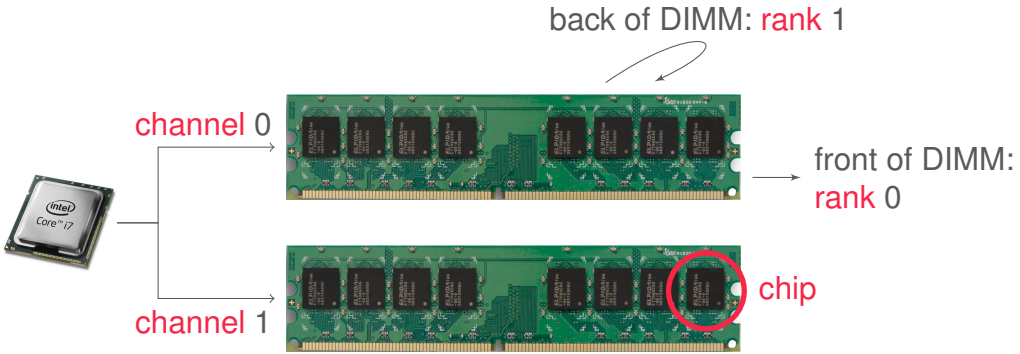
DRAM organization example



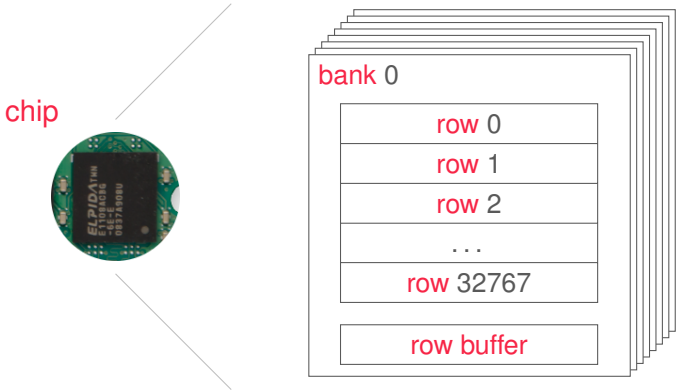
DRAM organization example



DRAM organization example



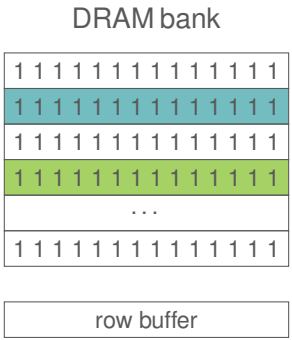
DRAM organization example



- bits in cells in rows
- access: activate row, copy to row buffer
- cells leak $\rightarrow$ refresh necessary
- cells leak faster upon proximate accesses

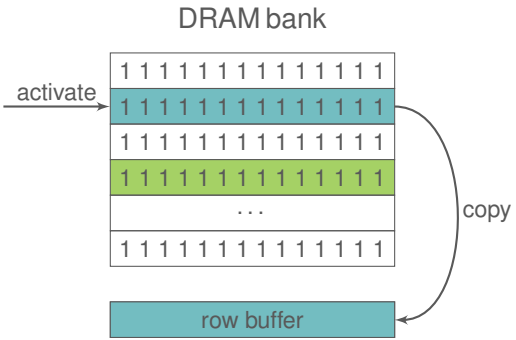
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice



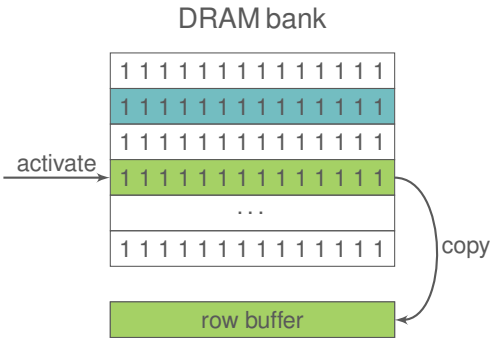
Rowhammer

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after" – Motherboard Vice



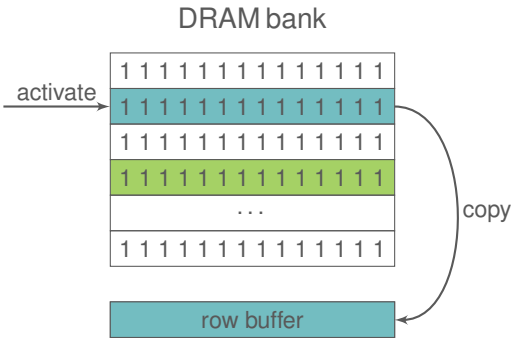
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice



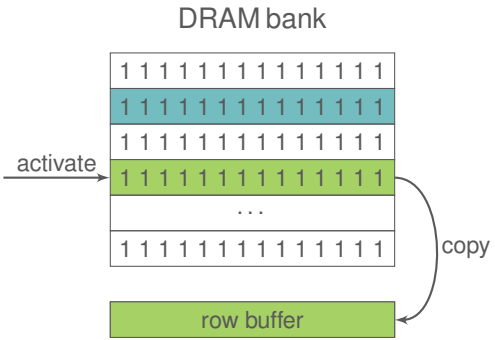
Rowhammer

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after" – Motherboard Vice



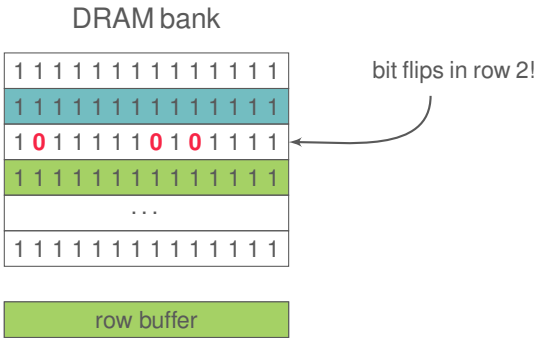
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

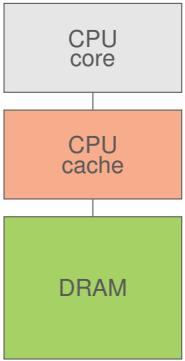


Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice



Impact of the CPU cache



- only non-cached accesses reach DRAM
- original attacks use `clflush` instruction

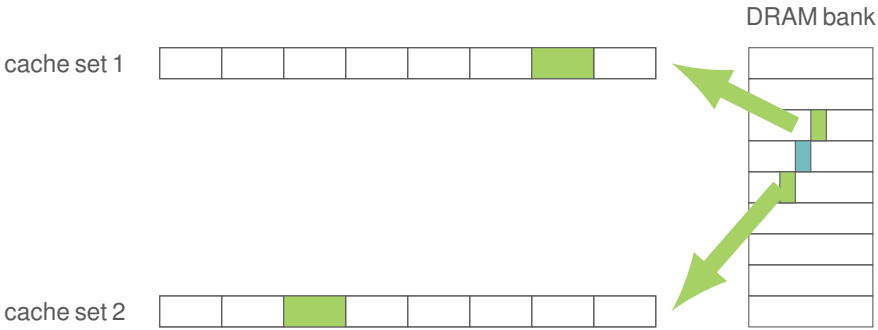
→ flush line from cache

→ next access will be served from DRAM

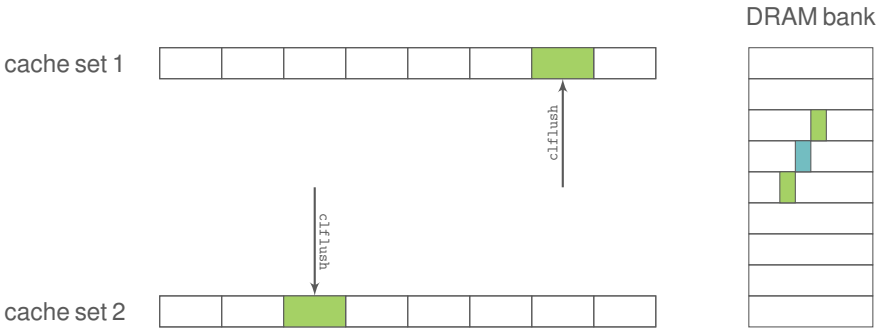
Rowhammer

- Rowhammer: DRAM bug that causes bit flips [Kim et al., 2014]
- Bug used in security exploits [Seaborn, 2015]
- Need non-cached accesses to reach DRAM
- Very similar to Flush+Reload

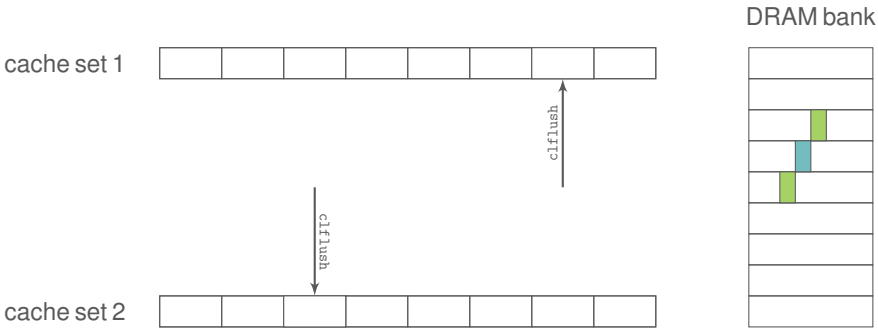
Rowhammer (with `clflush`)



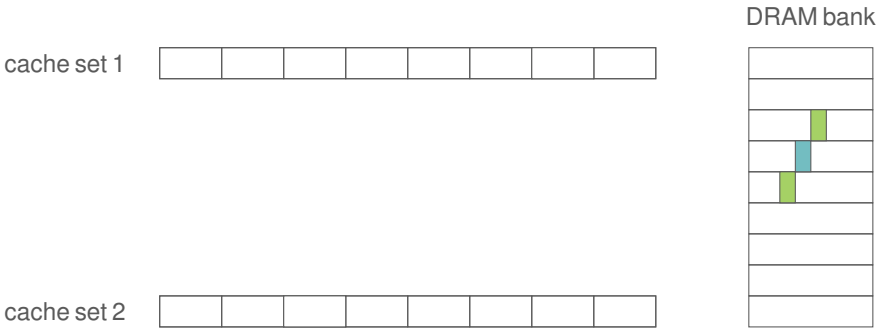
Rowhammer (with `clflush`)



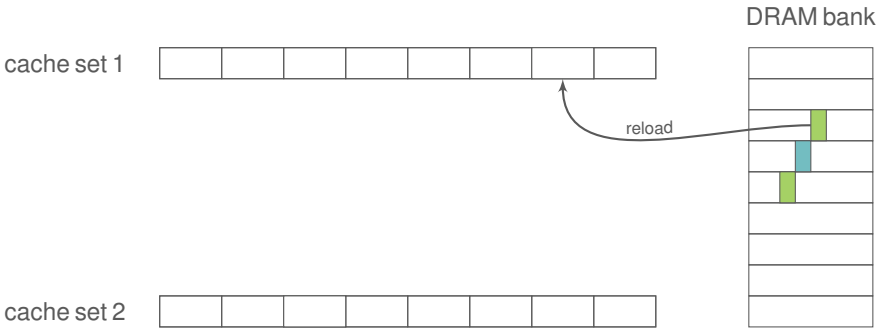
Rowhammer (with `clflush`)



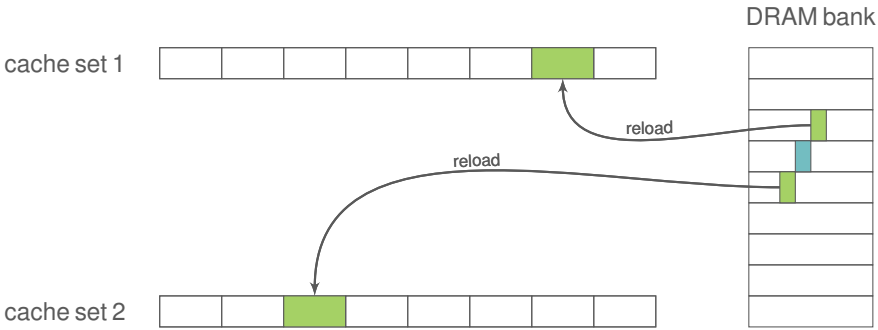
Rowhammer (with `clflush`)



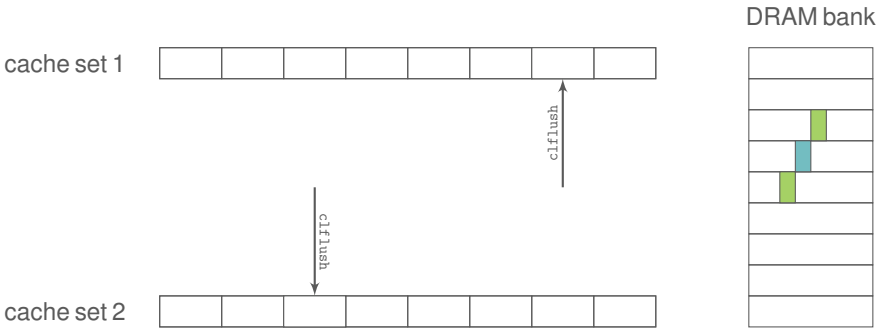
Rowhammer (with `clflush`)



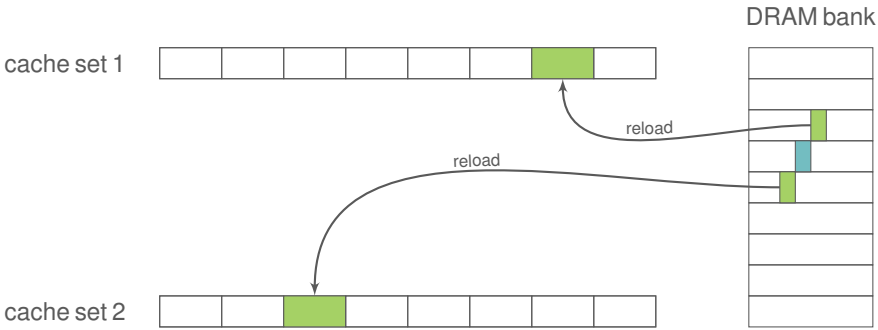
Rowhammer (with `clflush`)



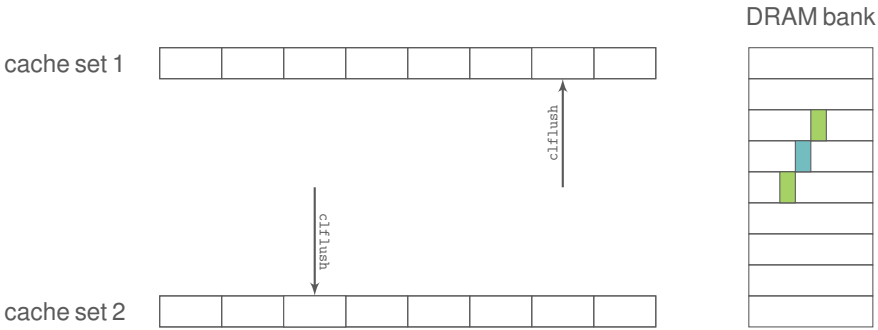
Rowhammer (with `clflush`)



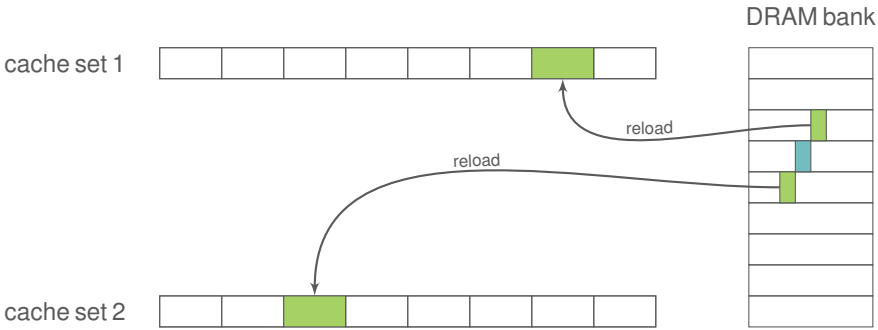
Rowhammer (with `clflush`)



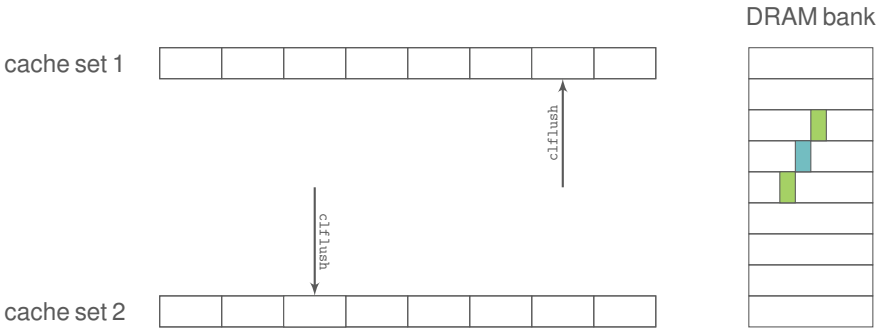
Rowhammer (with `clflush`)



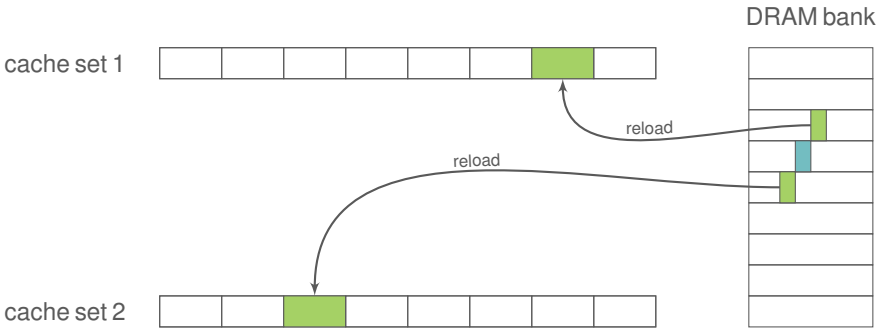
Rowhammer (with `clflush`)



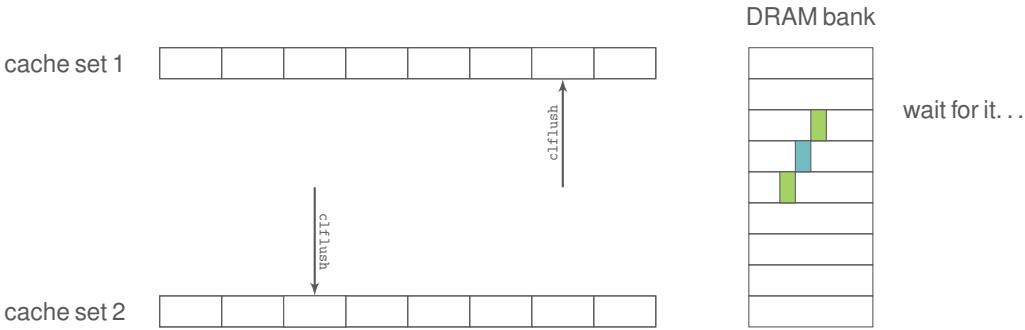
Rowhammer (with `clflush`)



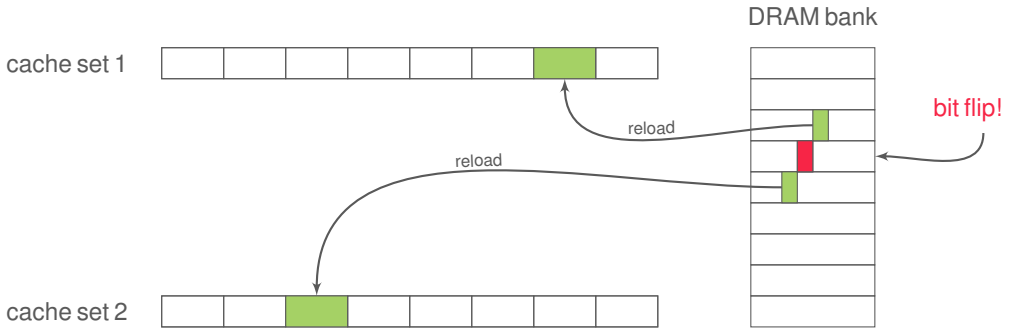
Rowhammer (with `clflush`)



Rowhammer (with `clflush`)



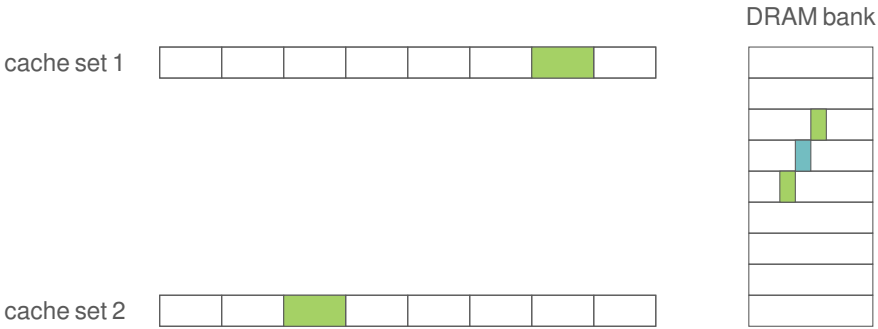
Rowhammer (with `clflush`)



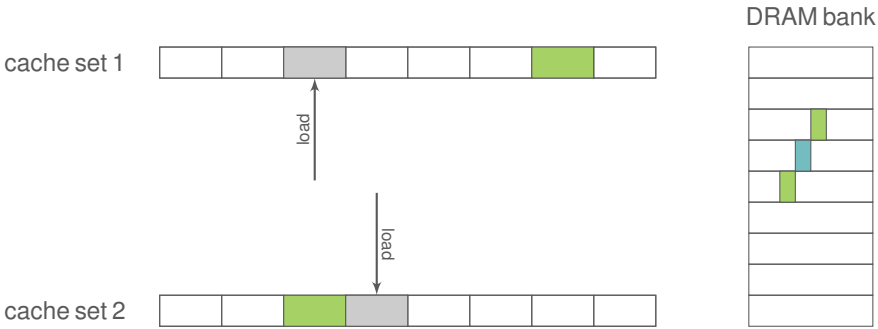
Rowhammer without `clflush`?

Idea: use Prime+Probe for Rowhammer instead of Flush+Reload

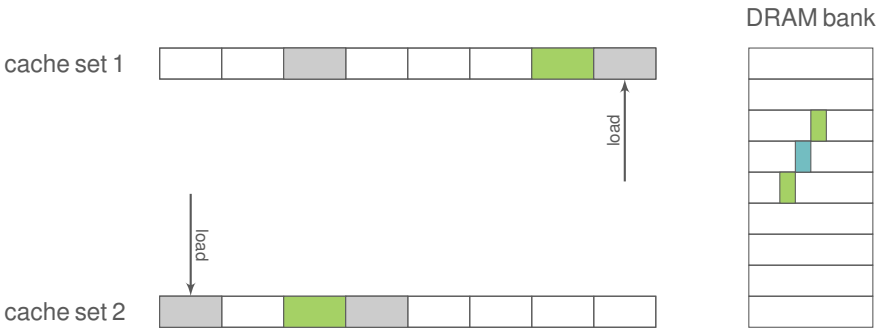
Rowhammer without `clflush`



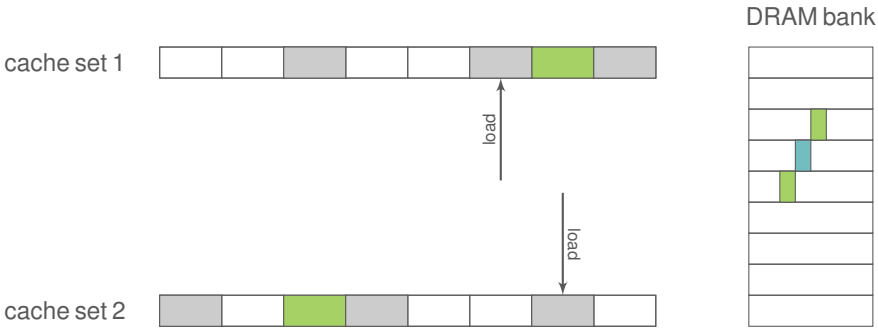
Rowhammer without `clflush`



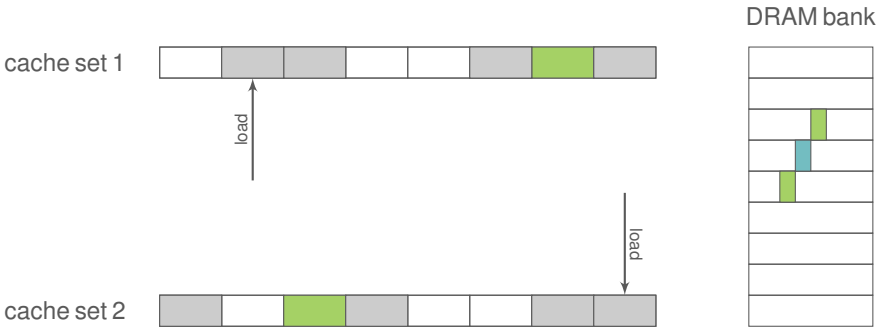
Rowhammer without `clflush`



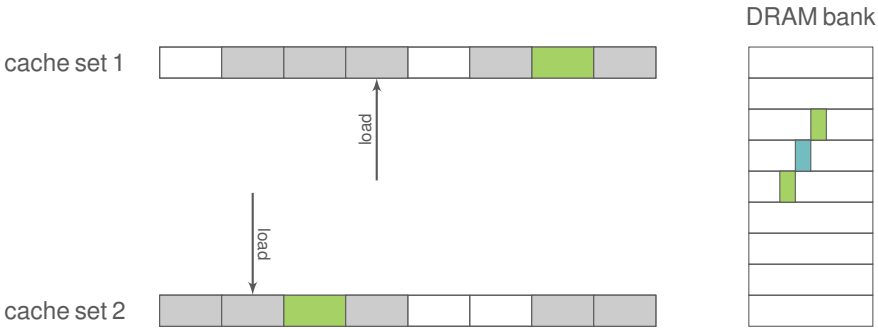
Rowhammer without `clflush`



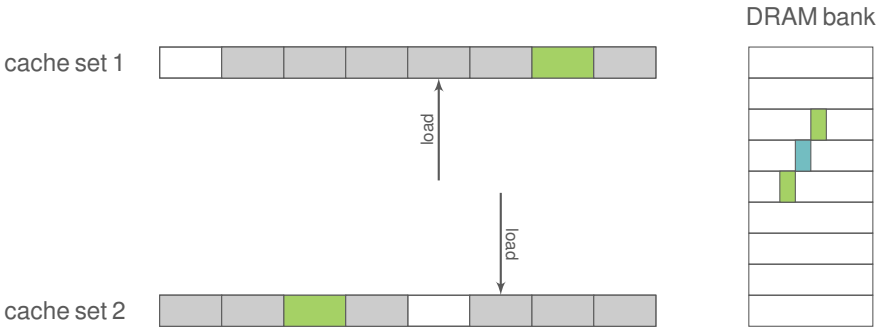
Rowhammer without `clflush`



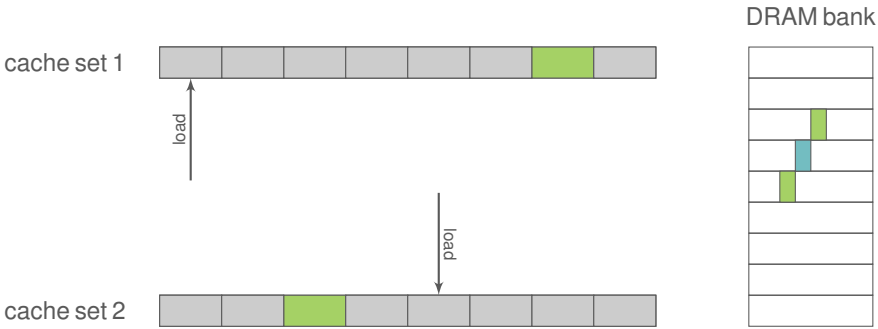
Rowhammer without `clflush`



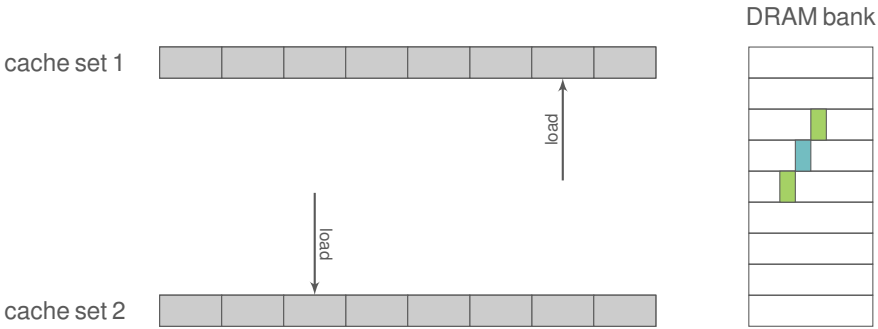
Rowhammer without `clflush`



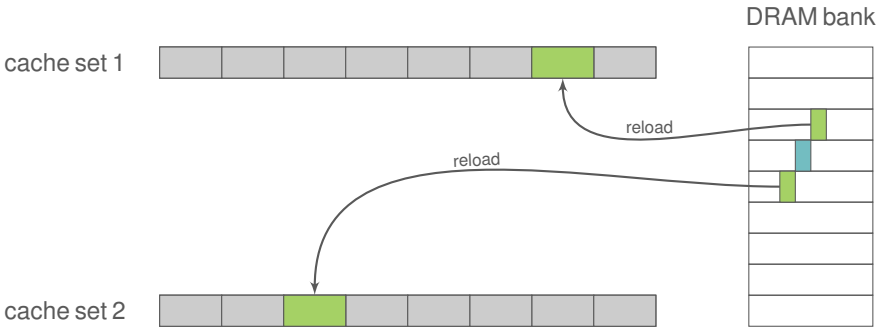
Rowhammer without `clflush`



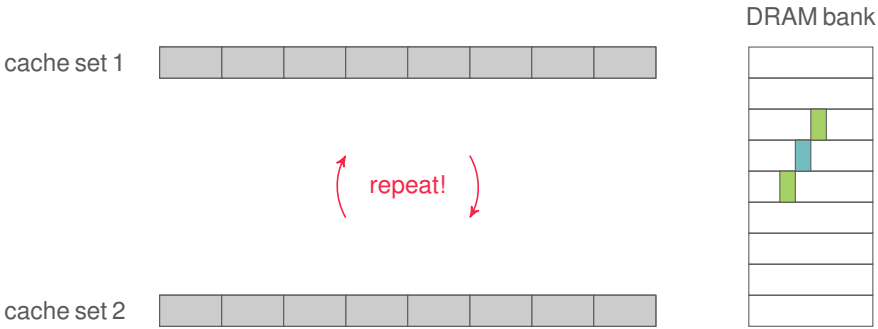
Rowhammer without `clflush`



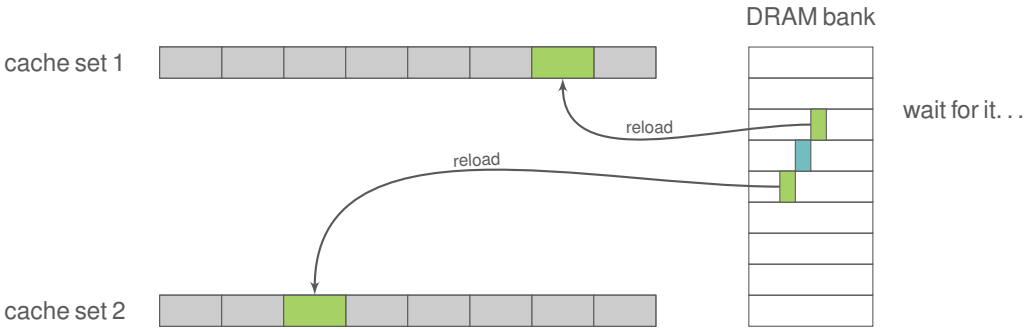
Rowhammer without `clflush`



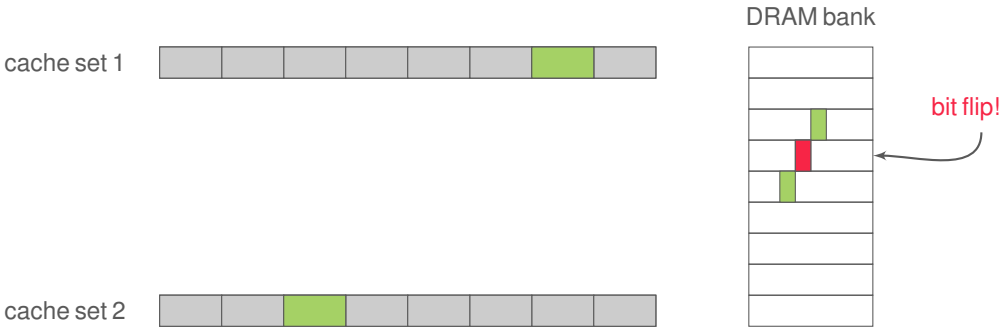
Rowhammer without `clflush`



Rowhammer without `clflush`



Rowhammer without `clflush`



Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)?
2. How to get physical addresses (in JS)?
3. Which physical addresses to access?
4. In which order to access them?

Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? $\rightarrow$ easy
2. How to get physical addresses (in JS)? $\rightarrow$ easy
3. Which physical addresses to access? $\rightarrow$ already solved
4. In which order to access them? $\rightarrow$ details today!

Accurate timing in JS (easy)

- native code: `rdtsc`
- JavaScript: `window.performance.now()`

Physical addresses in JS (easy)

- fixed map between physical addresses and DRAM
- we reverse-engineered it [Pessl et al., 2015]
- OS optimization: use 2MB pages
- last 21 bits equal for physical address, virtual address, JS array index [Gruss et al., 2015a]
- several DRAM rows per 2MB page

Which addresses to access? (already solved)

- use eviction $\rightarrow$ congruent addresses
- slice mapping is known [Maurice et al., 2015a]

Which addresses to access? (already solved)

- use eviction $\rightarrow$ congruent addresses
- slice mapping is known [Maurice et al., 2015a]

Remaining challenge: “in which order to access them?”

Replacement policy on older CPUs

“LRU eviction” memory accesses

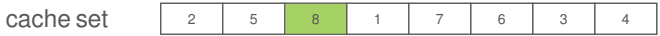
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first

Replacement policy on older CPUs

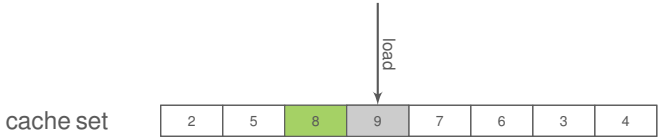
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line

Replacement policy on older CPUs

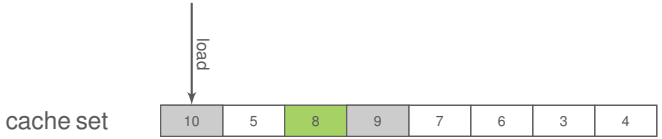
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

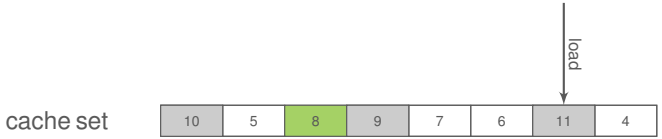
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

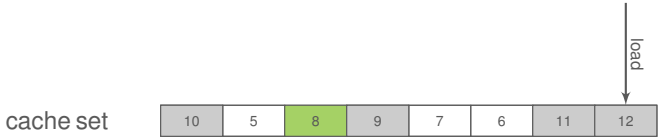
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

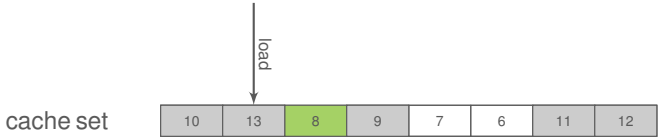
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

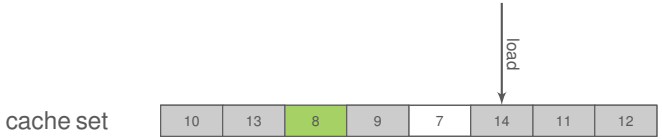
"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

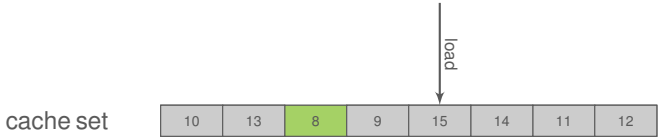
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

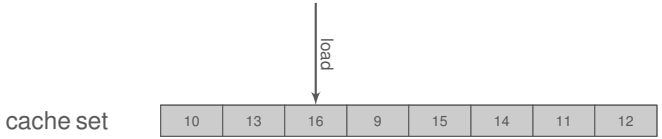
“LRU eviction” memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on older CPUs

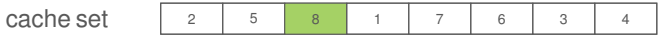
"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Replacement policy on recent CPUs

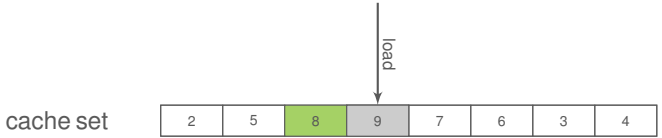
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

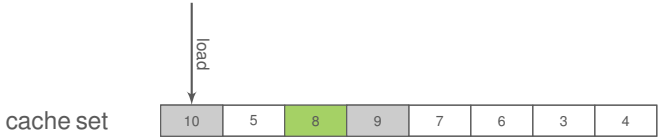
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

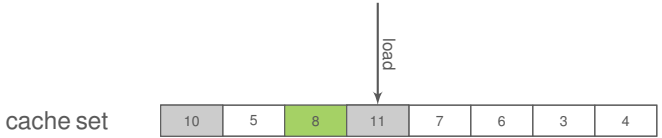
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

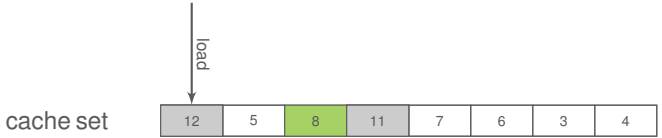
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

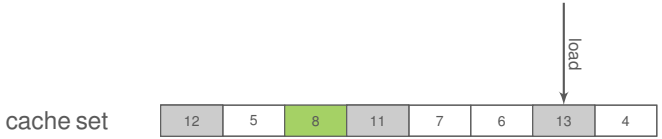
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

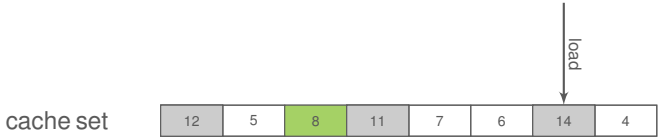
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

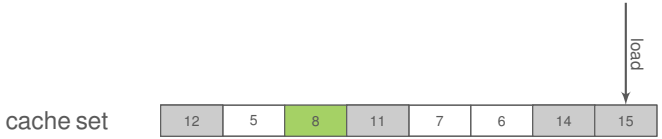
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

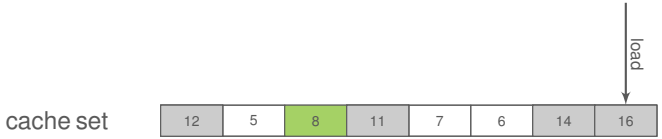
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

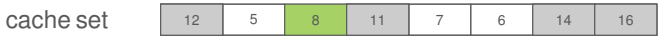
“LRU eviction” memory accesses



- no LRU replacement

Replacement policy on recent CPUs

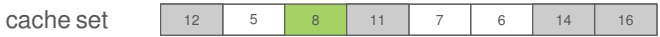
“LRU eviction” memory accesses



- no LRU replacement
- only 75% success rate on Haswell

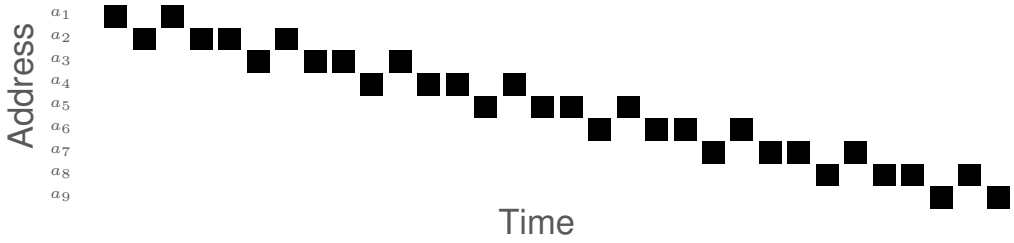
Replacement policy on recent CPUs

“LRU eviction” memory accesses



- no LRU replacement
- only 75% success rate on Haswell
- more accesses $\rightarrow$ higher success rate, but too slow

Cache eviction strategy (the beginning)



Cache eviction strategy (new representation)

Represent accesses as a sequence of numbers:

- $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, ...$
- can be a long sequence

Cache eviction strategy properties

- all addresses are congruent

$\rightarrow$ indistinguishable in terms of the eviction strategy

Cache eviction strategy properties

- all addresses are congruent

$\rightarrow$ indistinguishable in terms of the eviction strategy

- $2, 5, 4, 6, 3, 1$, $2, 5, 6, 4, 1, 3$, ... all equivalent!

$\rightarrow$ $1, 2, 3, 4, 5, 6$

Cache eviction strategy properties

- all addresses are congruent

$\rightarrow$ indistinguishable in terms of the eviction strategy

  - $2, 5, 4, 6, 3, 1, 2, 5, 6, 4, 1, 3$, ... all equivalent!

  $\rightarrow$ $1, 2, 3, 4, 5, 6$

- adding more different addresses can increase eviction rate
- multiple accesses to one address can increase the eviction rate

Cache eviction strategy properties (2)

- multiple accesses, low number of different addresses

$\rightarrow$ sequences often have repeating sub-sequences

Cache eviction strategy properties (2)

- multiple accesses, low number of different addresses

$\rightarrow$ sequences often have repeating sub-sequences

- idea: repetative sequences can be compressed to loops over sub-sequences

Cache eviction strategy (compression)

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy (compression)

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

D: different addresses per inner access loop

C: number of repetitions of the inner access loop

S: total number of different addresses (=set size)

L: overlap parameter for the inner access loops (=step size)

Cache eviction strategy (compression)

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

D: different addresses per inner access loop

C: number of repetitions of the inner access loop

S: total number of different addresses (=set size)

L: overlap parameter for the inner access loops (=step size)

Write eviction strategies as: $\mathcal{P}$-C-D-L-S

Cache eviction strategy example

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c < C; c += 1)
    for (d = 0; d < D; d += 1)
      *a[s+d];
```

Cache eviction strategy example

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c < C; c += 1)
    for (d = 0; d < D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-1-1-1-4
- $\mathcal{P}$-2-2-1-4

Cache eviction strategy example

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c < C; c += 1)
    for (d = 0; d < D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-1-1-1-4 $\rightarrow 1, 2, 3, 4$
- $\mathcal{P}$-2-2-1-4 $\rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$

Cache eviction strategy example

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c < C; c += 1)
    for (d = 0; d < D; d += 1)
      *a[s+d];
```

- $\mathcal{P}$-1-1-1-4 $\rightarrow 1, 2, 3, 4$
- $\mathcal{P}$-2-2-1-4 $\rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$

We evaluated more than 10000 strategies...

Cache eviction strategies (simple example)

Which one has the better eviction rate?

- $\mathcal{P}$-1-1-1-17 (as used in previous work)
- $\mathcal{P}$-1-1-1-20

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (simple example)

Which one has the better eviction rate?

- $\mathcal{P}$-1-1-1-17 (as used in previous work)
  - $\rightarrow$ Average eviction rate: 74.46%
  - $\rightarrow$ Not high enough for Rowhammer!
- $\mathcal{P}$-1-1-1-20
  - $\rightarrow$ Average eviction rate: 99.82%
  - $\rightarrow$ High enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (simple example)

Which one is faster?

- $\mathcal{P}$-1-1-1-17 (as used in previous work)
- $\mathcal{P}$-1-1-1-20 (=20 accesses)

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (simple example)

Which one is faster?

- $\mathcal{P}$-1-1-1-17 (as used in previous work)
  - $\rightarrow$ Average execution time: 307 ns
  - $\rightarrow$ Fast enough for Rowhammer!
- $\mathcal{P}$-1-1-1-20 (=20 accesses)
  - $\rightarrow$ Average execution time: 934 ns
  - $\rightarrow$ Not fast enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}$-1-1-1-17 (=17 accesses)

  $\rightarrow$ Average eviction rate: 74.46%

- $\mathcal{P}$-2-1-1-17 (=34 accesses)

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}$-1-1-1-17 (=17 accesses)
  - $\rightarrow$ Average eviction rate: 74.46%
- $\mathcal{P}$-2-1-1-17 (=34 accesses)
  - $\rightarrow$ Average eviction rate: 99.86%
  - $\rightarrow$ High enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one is faster?

- $\mathcal{P}$-1-1-1-17 (=17 accesses)

  $\rightarrow$ Average execution time: 307 ns

- $\mathcal{P}$-2-1-1-17 (=34 accesses)

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one is faster?

- $\mathcal{P}$-1-1-1-17 (=17 accesses)
  - $\rightarrow$ Average execution time: 307 ns
- $\mathcal{P}$-2-1-1-17 (=34 accesses)
  - $\rightarrow$ Average execution time: 191 ns
  - $\rightarrow$ Fast enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}$-2-1-1-17 (=34 accesses)

  $\rightarrow$ Average eviction rate: 99.86%
  $\rightarrow$ Average execution time: 191 ns

- $\mathcal{P}$-2-2-1-17 (=64 accesses)

(executed in a loop, on Haswell, 16-way cache)

Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}$-2-1-1-17 (=34 accesses)
  - $\rightarrow$ Average eviction rate: 99.86%
  - $\rightarrow$ Average execution time: 191 ns
- $\mathcal{P}$-2-2-1-17 (=64 accesses)
  - $\rightarrow$ Average eviction rate: 99.98%
  - $\rightarrow$ Average execution time: 180 ns

(executed in a loop, on Haswell, 16-way cache)

Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? $\rightarrow$ easy
2. How to get physical addresses (in JS)? $\rightarrow$ easy
3. Which physical addresses to access? $\rightarrow$ solved
4. In which order to access them?

Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? $\rightarrow$ easy
2. How to get physical addresses (in JS)? $\rightarrow$ easy
3. Which physical addresses to access? $\rightarrow$ solved
4. In which order to access them? $\rightarrow$ solved

Evaluation on Haswell

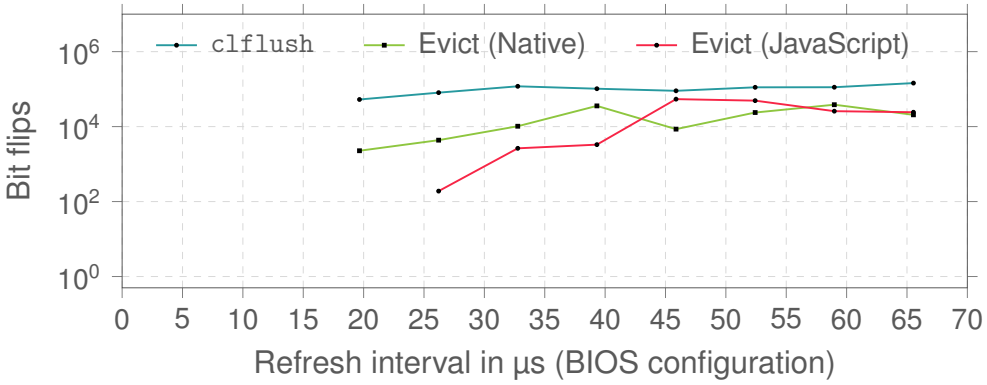


Figure: Number of bit flips within 15 minutes. [Gruss et al., 2016]

Conclusions

- cache eviction good enough to replace `clflush`
- independent of programming language and available instructions
- hardware-fault attack induced in JavaScript

$\rightarrow$ remote attacks through websites are possible

At least on this laptop (default settings)

ROOT privileges for web apps!

Cache Side-Channel Attacks and the case of Rowhammer

Daniel Gruss
IAIK, Graz University of Technology

April 28, 2016

[Gruss et al., 2015a] Gruss, D., Bidner, D., and Mangard, S. (2015a).

Practical Memory Deduplication Attacks in Sandboxed JavaScript.

In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)*.

[Gruss et al., 2016] Gruss, D., Maurice, C., and Mangard, S. (2016).

Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript.

In *DIMVA'16*.

[Gruss et al., 2015b] Gruss, D., Spreitzer, R., and Mangard, S. (2015b).

Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.

In *USENIX Security Symposium (USENIX Security'15)*.

[Gullasch et al., 2011] Gullasch, D., Bangerter, E., and Krenn, S. (2011).

Cache Games – Bringing Access-Based Cache Attacks on AES to Practice.

In *IEEE Symposium on Security and Privacy (S&P'11)*.

[Hund et al., 2013] Hund, R., Willems, C., and Holz, T. (2013).

Practical Timing Side Channel Attacks against Kernel Space ASLR.

In *IEEE Symposium on Security and Privacy – S&P*, pages 191–205. IEEE.

[Inci et al., 2015] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. (2015).

Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.

Cryptology ePrint Archive, Report 2015/898.

[Kim et al., 2014] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. (2014).

Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.

In *ACM/IEEE International Symposium on Computer Architecture (ISCA'14)*.

[Liu et al., 2015] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015).

Last-Level Cache Side-Channel Attacks are Practical.

In *IEEE Symposium on Security and Privacy (S&P'15)*.

[Maurice et al., 2015a] Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. (2015a).

Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters.

In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[Maurice et al., 2015b] Maurice, C., Neumann, C., Heen, O., and Francillon, A. (2015b).

C5: Cross-Cores Cache Covert Channel.

In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'15)*.

[Percival, 2005] Percival, C. (2005).

Cache Missing for Fun and Profit.

URL: `http://daemonology.net/hyperthreading-considered-harmful/`.

[Pessl et al., 2015] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. (2015).

Reverse Engineering Intel DRAM Addressing and Exploitation.

arXiv:1511.08756.

[Seaborn, 2015] Seaborn, M. (2015).

Exploiting the DRAM rowhammer bug to gain kernel privileges.

http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[Yarom and Falkner, 2014] Yarom, Y. and Falkner, K. (2014).

FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.

In *USENIX Security Symposium (USENIX Security'14)*.

[Yarom et al., 2015] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. (2015).

Mapping the Intel Last-Level Cache.

Cryptology ePrint Archive, Report 2015/905.