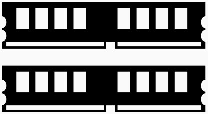
# Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features

**Michael Schwarz**, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, Stefan Mangard

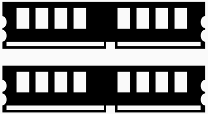
June 7, 2018

# Double Fetches



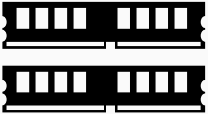
- A time-of-check-to-time-of-use (TOCTTOU) bug

# Double Fetches



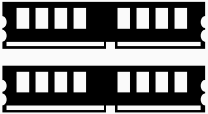
- A time-of-check-to-time-of-use (TOCTTOU) bug
- Shared memory might change after sanity check

# Double Fetches



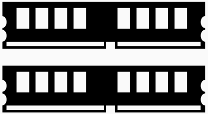
- A time-of-check-to-time-of-use (TOCTTOU) bug
- Shared memory might change after sanity check
- Adversary can abuse this to provide invalid data to application

# Double Fetches



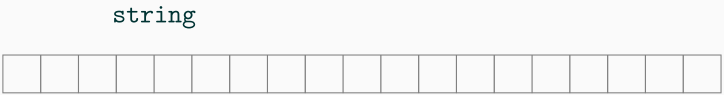
- A time-of-check-to-time-of-use (TOCTTOU) bug
- Shared memory might change after sanity check
- Adversary can abuse this to provide invalid data to application
- Caused by accessing the shared memory twice

# Double Fetches

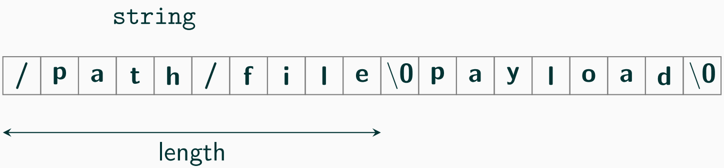


- A time-of-check-to-time-of-use (TOCTTOU) bug
- Shared memory might change after sanity check
- Adversary can abuse this to provide invalid data to application
- Caused by accessing the shared memory twice
- Also called double-fetch bugs

# A Double Fetch



# A Double Fetch

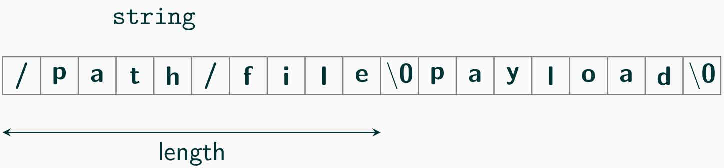


Thread 1

```
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);
```

Thread 2

# A Double Fetch



string

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| / | p | a | t | h | / | f | i | l | e | \0 | p | a | y | l | o | a | d | \0 |

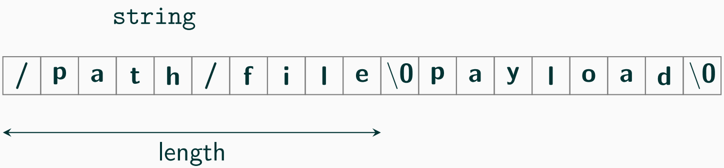
length

Thread 1

Thread 2

```
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);

// <switch to kernel>
```

# A Double Fetch

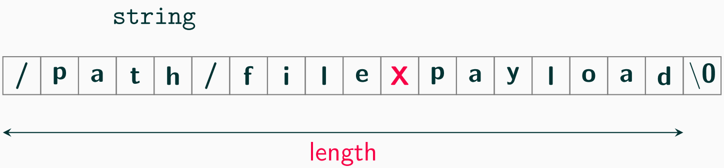


Thread 1

Thread 2

```
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);

// <switch to kernel>

int len = strlen(string);
char* local = malloc(len + 1);
```

# A Double Fetch



Thread 1

```
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);

// <switch to kernel>

int len = strlen(string);
char* local = malloc(len + 1);
```

schedule

Thread 2

```
string[10] = 'X';
```

# A Double Fetch

string

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| / | p | a | t | h | / | f | i | l | e | X | p | a | y | l | o | a | d | \0 |

length

Thread 1

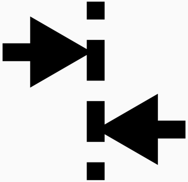
```
strcpy(string, "/path/file\0payload");
open(string, O_CREAT);

// <switch to kernel>

int len = strlen(string);
char* local = malloc(len + 1);

strcpy(local, string);
// <memory corruption>
```

Thread 2

schedule

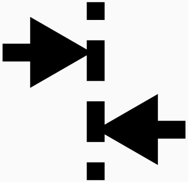
```
string[10] = 'X';
```

# Double Fetches



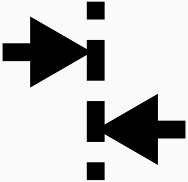
- Not all double fetches are exploitable

# Double Fetches



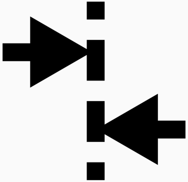
- Not all double fetches are exploitable
- Changing data after sanity check allows exploitation

# Double Fetches



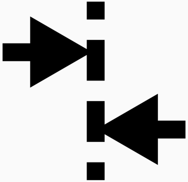
- Not all double fetches are exploitable
- Changing data after sanity check allows exploitation
- Critical if privilege boundaries are crossed

# Double Fetches



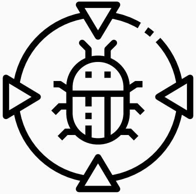
- Not all double fetches are exploitable
- Changing data after sanity check allows exploitation
- Critical if privilege boundaries are crossed
    - User space ↔ Kernel space
    - Untrusted code ↔ Trusted code

# Double Fetches



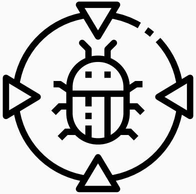
- Not all double fetches are exploitable
- Changing data after sanity check allows exploitation
- Critical if privilege boundaries are crossed
  - User space ↔ Kernel space
  - Untrusted code ↔ Trusted code
- Common to share data across these domains

# Detecting Double Fetches



- Race condition → hard to detect

# Detecting Double Fetches

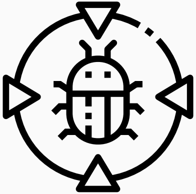


- Race condition → hard to detect
- Sometimes code is not available → secure enclaves

# Detecting Double Fetches

- Race condition → hard to detect
- Sometimes code is not available → secure enclaves
- Static code analysis [WK17] not applicable

# Detecting Double Fetches

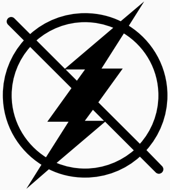


- Race condition → hard to detect
- Sometimes code is not available → secure enclaves
- Static code analysis [WK17] not applicable
- Emulation [J+13] is slow and for enclaves not applicable

# DECAF

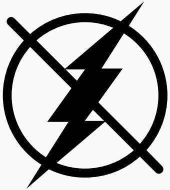
## Double-fetch-exposing Cache-guided Augmentation for Fuzzers

# DECAF



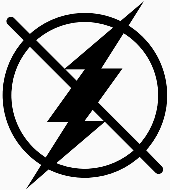
- A dynamic approach based on fuzzing and cache attacks

# DECAF



- A dynamic approach based on fuzzing and cache attacks
- Basic idea: memory access can be observed in the cache

# DECAF

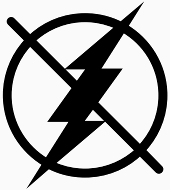


- A dynamic approach based on fuzzing and cache attacks
- Basic idea: memory access can be observed in the cache
- Observe cache activity using a cache attack

# DECAF

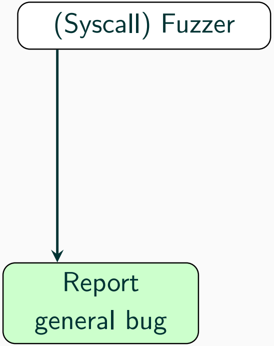
- A dynamic approach based on fuzzing and cache attacks
- Basic idea: memory access can be observed in the cache
- Observe cache activity using a cache attack
- Combine with fuzzing to cover many execution paths

# DECAF

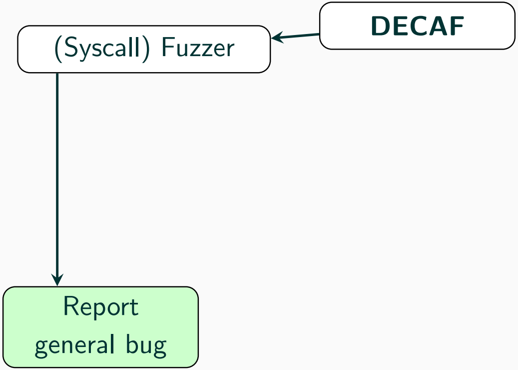


- A dynamic approach based on fuzzing and cache attacks
- Basic idea: memory access can be observed in the cache
- Observe cache activity using a cache attack
- Combine with fuzzing to cover many execution paths
- Automatically exploit double fetches to eliminate false positives

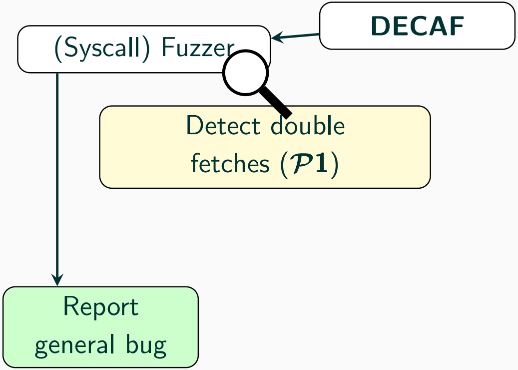
# DECAF



# DECAF



# DECAF

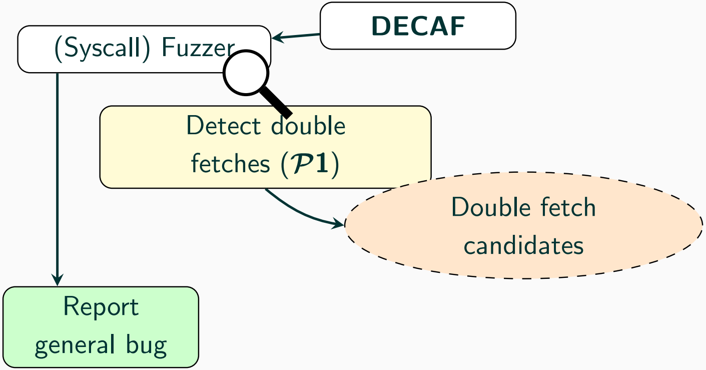


**DECAF**

(Syscall) Fuzzer

Detect double fetches ($\mathcal{P}\mathbf{1}$)

Report general bug



**DECAF**

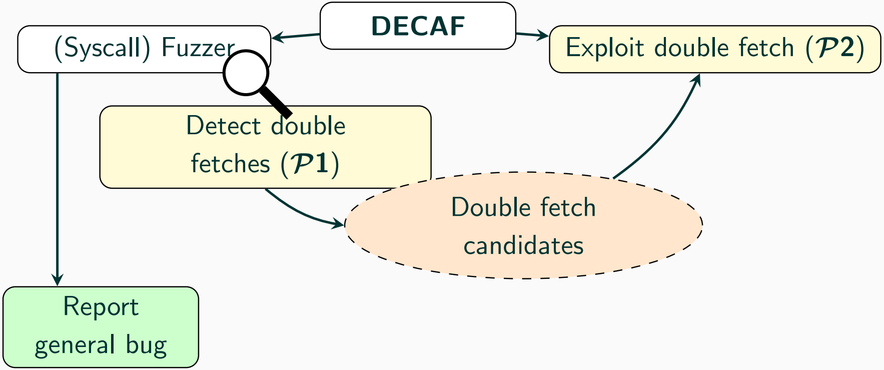
(Syscall) Fuzzer

Detect double fetches ($\mathcal{P}$**1**)

Double fetch candidates

Report general bug

# DECAF


**DECAF**

(Syscall) Fuzzer

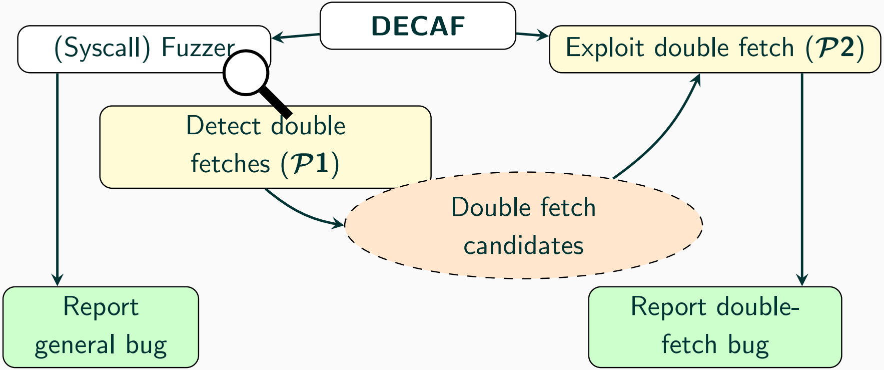
Exploit double fetch ($\mathcal{P}\mathbf{2}$)

Detect double fetches ($\mathcal{P}\mathbf{1}$)

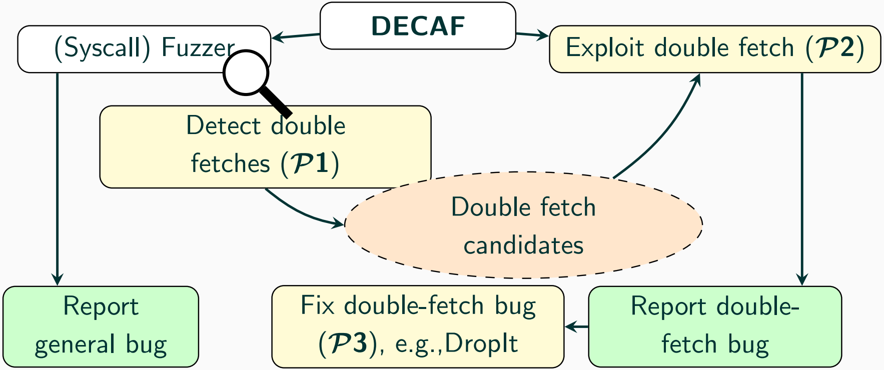
Double fetch candidates

Report general bug

# DECAF


DECAF
(Syscall) Fuzzer
Exploit double fetch ($\mathcal{P}\mathbf{2}$)
Detect double fetches ($\mathcal{P}\mathbf{1}$)
Double fetch candidates
Report general bug
Report double-fetch bug

# DECAF



**DECAF**

(Syscall) Fuzzer

Exploit double fetch ($\mathcal{P}\mathbf{2}$)

Detect double fetches ($\mathcal{P}\mathbf{1}$)

Double fetch candidates

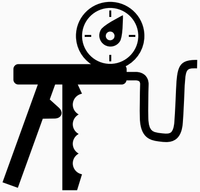
Report general bug

Fix double-fetch bug ($\mathcal{P}\mathbf{3}$), e.g.,DropIt

Report double-fetch bug

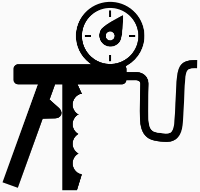
# $\mathcal{P}1$: Detection

# Flush+Reload



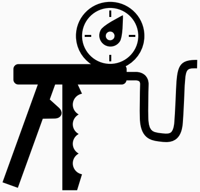
- Shared memory make cache observation easy

# Flush+Reload



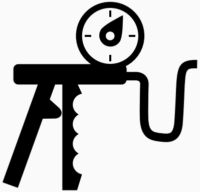
- Shared memory make cache observation easy
- Constantly flush value from cache using `clflush` instruction

# Flush+Reload



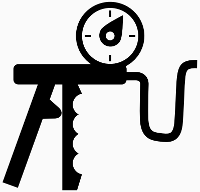
- Shared memory make cache observation easy
- Constantly flush value from cache using `clflush` instruction
- Schedule victim

# Flush+Reload



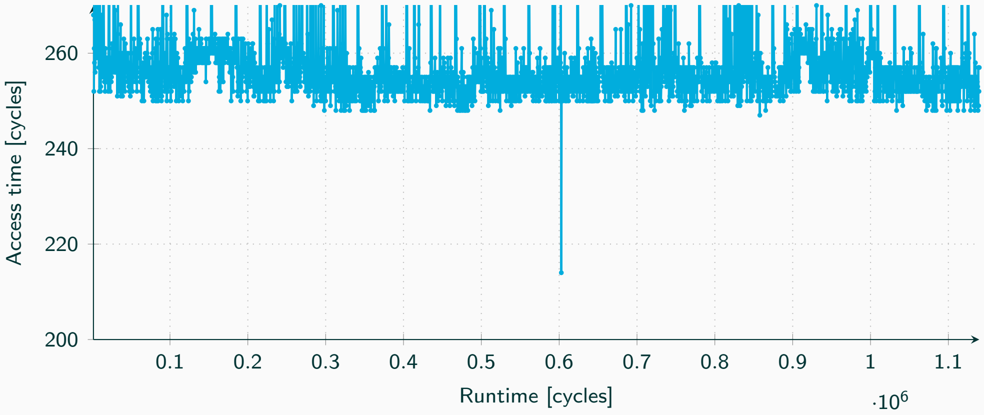
- Shared memory make cache observation easy
- Constantly flush value from cache using `clflush` instruction
- Schedule victim
- Measure access time to value

# Flush+Reload

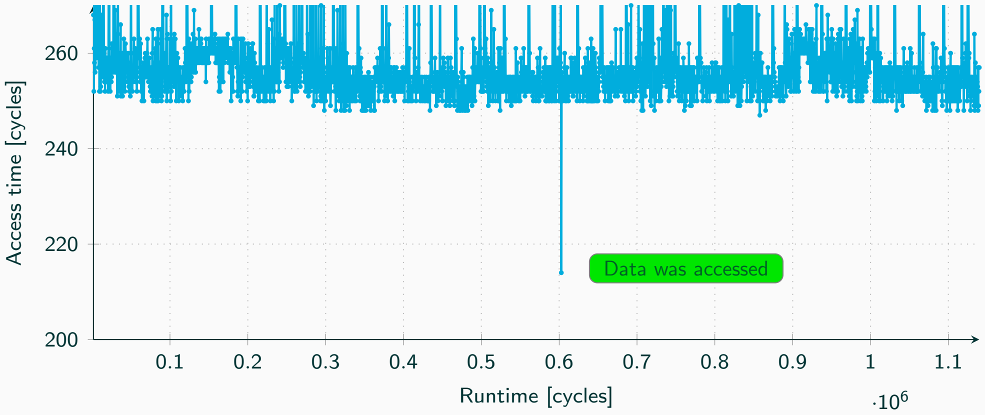


- Shared memory make cache observation easy
- Constantly flush value from cache using `clflush` instruction
- Schedule victim
- Measure access time to value
- If access is fast, victim accessed (and cached) the data

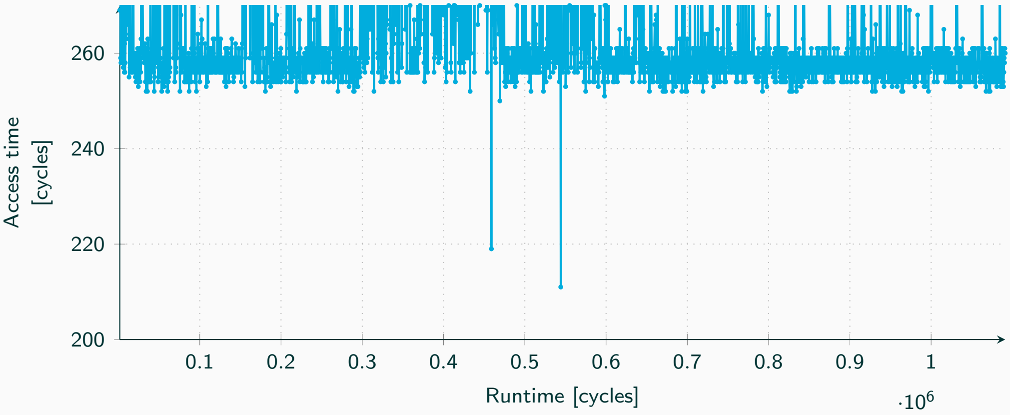
# Flush+Reload



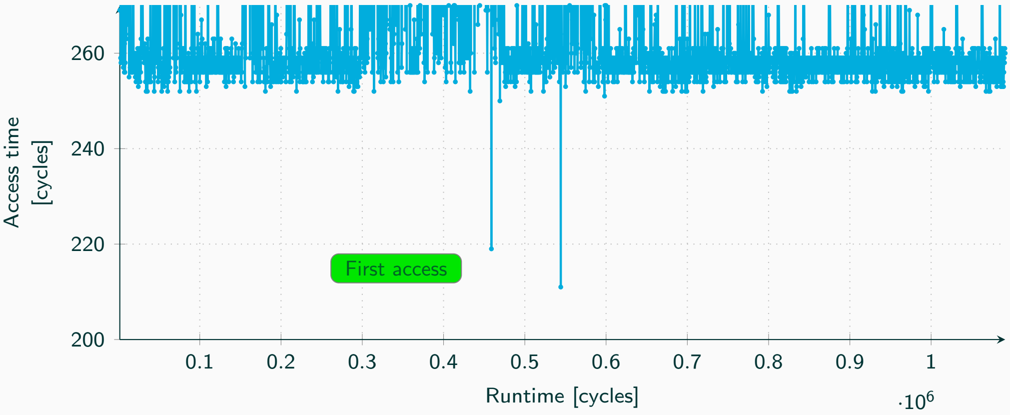
# Flush+Reload



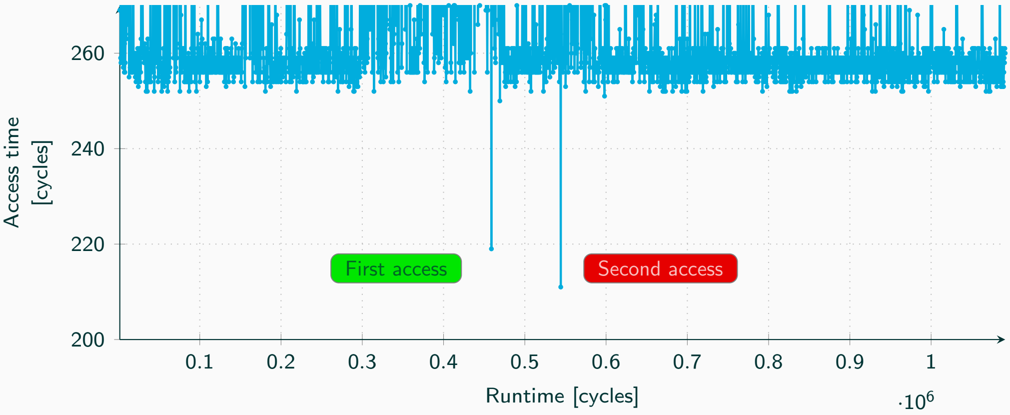
# Double-fetch Detection



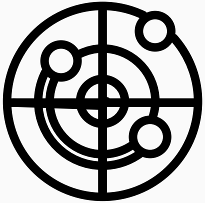
# Double-fetch Detection



# Double-fetch Detection

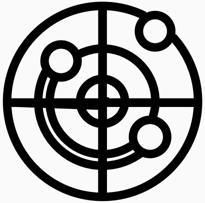


# Dynamic Double-fetch Detection



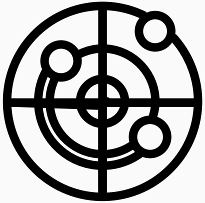
- Cache attacks dynamically detect double fetches

# Dynamic Double-fetch Detection



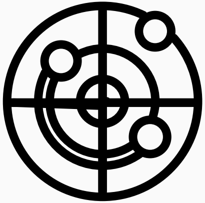
- Cache attacks dynamically detect double fetches
- The further apart the two fetches, the higher the probability

# Dynamic Double-fetch Detection



- Cache attacks dynamically detect double fetches
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles

# Dynamic Double-fetch Detection



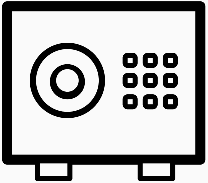
- Cache attacks dynamically detect double fetches
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles
- If the time is $\geq 3\,000$ cycles, detection rate is close to $100\,\%$

# Dynamic Double-fetch Detection

- Cache attacks dynamically detect double fetches
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles
- If the time is $\geq$ 3 000 cycles, detection rate is close to 100 %
- Allows automated detection of double fetches in many scenarios

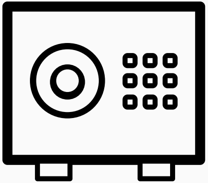
# Dynamic Double-fetch Detection

- Not limited to syscalls and not limited to operating system

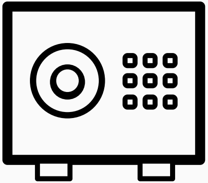


# Dynamic Double-fetch Detection

- Not limited to syscalls and not limited to operating system
- Oblivious to programming language and programming constructs

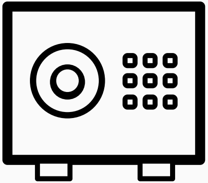


# Dynamic Double-fetch Detection

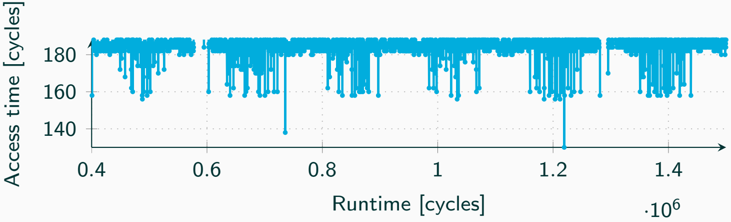


- Not limited to syscalls and not limited to operating system
- Oblivious to programming language and programming constructs
- For every black box if memory is shared

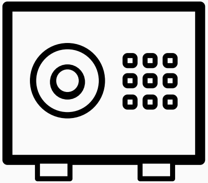
# Dynamic Double-fetch Detection



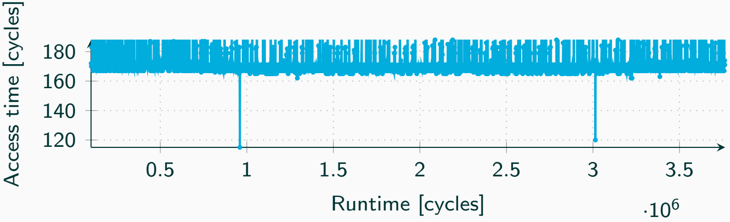
- Not limited to syscalls and not limited to operating system
- Oblivious to programming language and programming constructs
- For every black box if memory is shared
- Verified for SGX



# Dynamic Double-fetch Detection



- Not limited to syscalls and not limited to operating system
- Oblivious to programming language and programming constructs
- For every black box if memory is shared
- Verified for TrustZone



# $\mathcal{P}2$: Exploitation

# Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable

# Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable
- Double-fetch bug if exploitable

# Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable
- Double-fetch bug if exploitable
- Only double-fetch bugs are interesting

# Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable
- Double-fetch bug if exploitable
- Only double-fetch bugs are interesting
- Automatically try to exploit while fuzzing

# Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable
- Double-fetch bug if exploitable
- Only double-fetch bugs are interesting
- Automatically try to exploit while fuzzing
- Distinguish double fetches from double-fetch bugs

# Double-fetch Bug Exploitation

- Only a small time window to modify the value

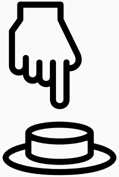
# Double-fetch Bug Exploitation

- Only a small time window to modify the value
- State-of-the-art exploitation:
  - Flip the value as fast as possible between two values
  - At some point, we hit the time window
  - Probability is not very high

# Double-fetch Bug Exploitation

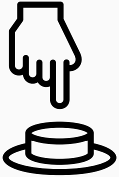
- Only a small time window to modify the value
- State-of-the-art exploitation:
  - Flip the value as fast as possible between two values
  - At some point, we hit the time window
  - Probability is not very high
- Use a trigger

# Cache-based Trigger



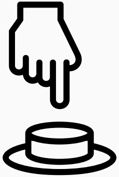
- Use the cache side channel as trigger

# Cache-based Trigger



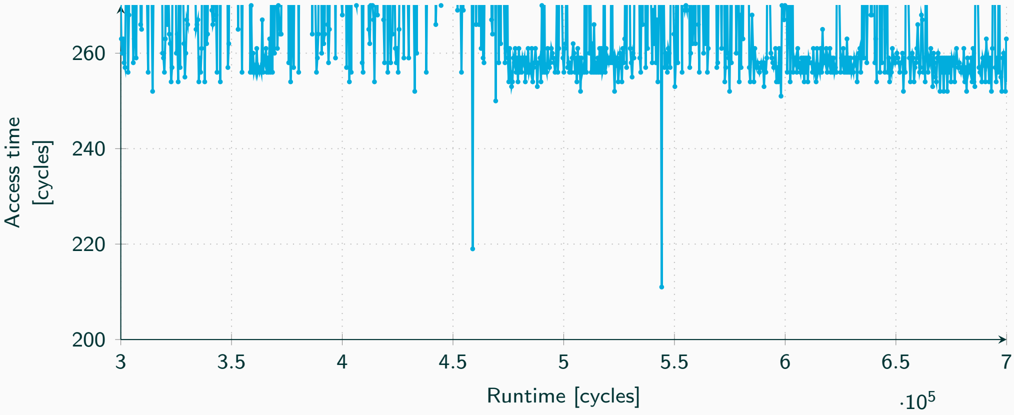
- Use the cache side channel as trigger
- The first memory access acts as trigger

# Cache-based Trigger

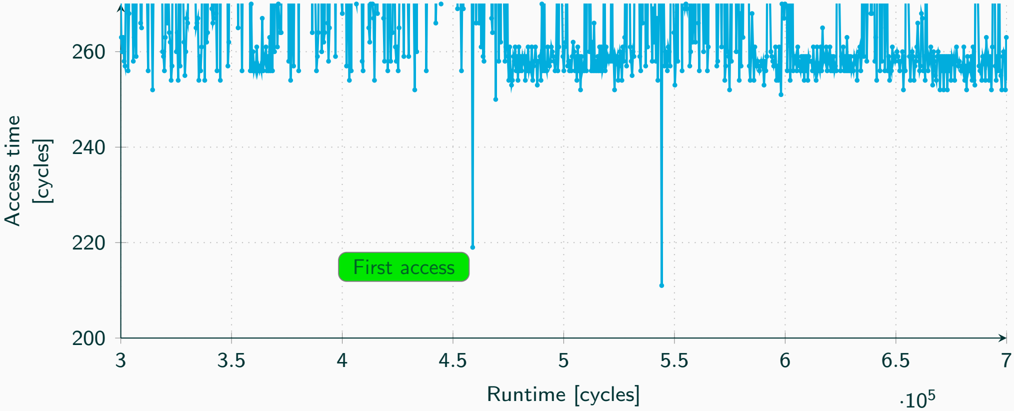


- Use the cache side channel as trigger
- The first memory access acts as trigger
- Simply change the value immediately after the first access

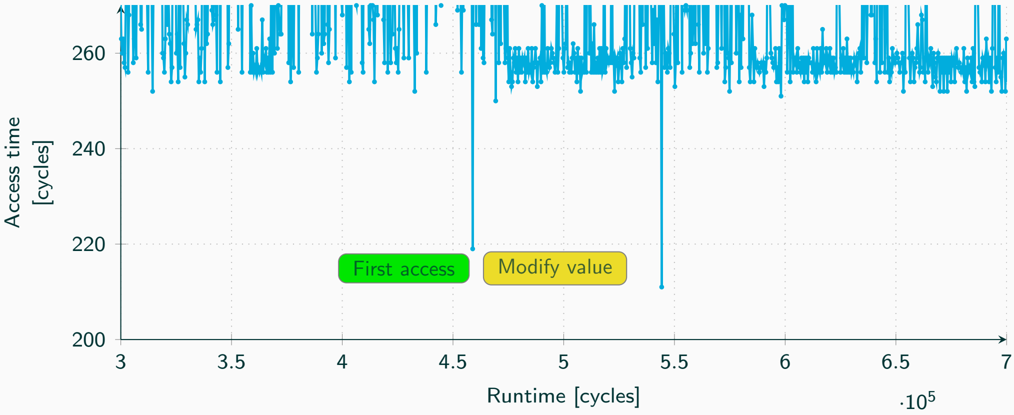
# Cache-based Trigger



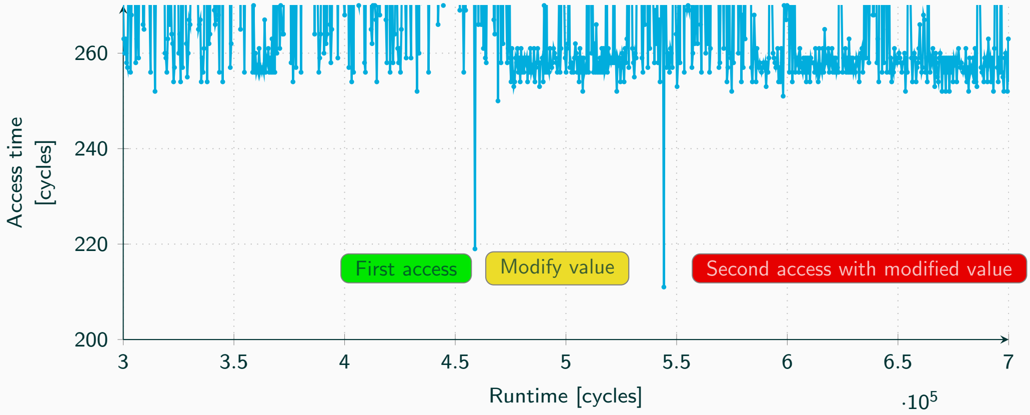
# Cache-based Trigger



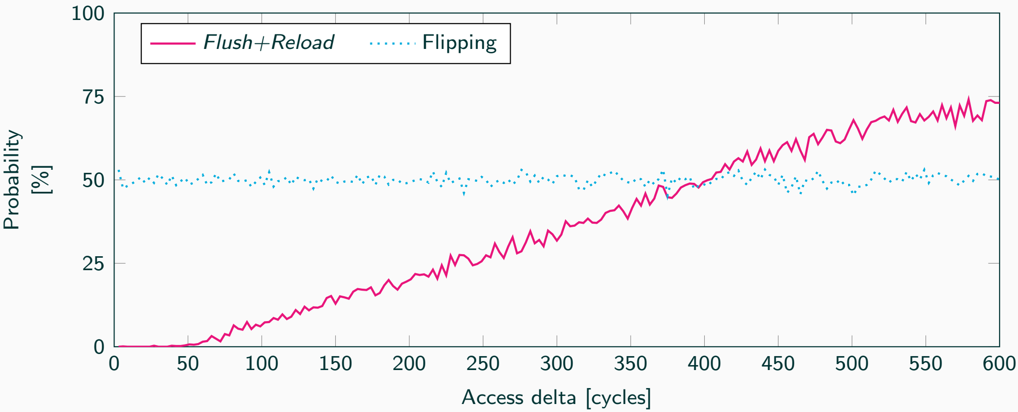
# Cache-based Trigger



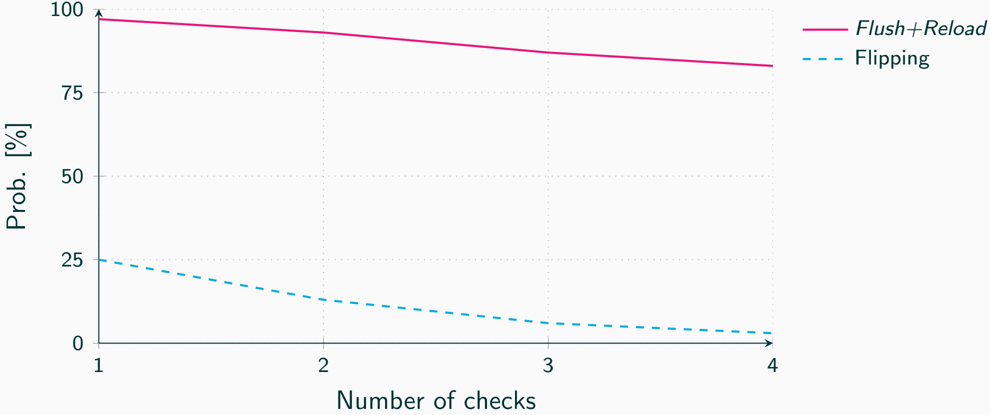
# Cache-based Trigger



# Cache-based Trigger



# Cache-based Trigger



# $\mathcal{P}$3: Prevention

# Eliminate Double-fetch Bugs

- Double fetches are not bad by itself

# Eliminate Double-fetch Bugs

- Double fetches are not bad by itself
- Sometimes they are necessary

# Eliminate Double-fetch Bugs

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit

# Eliminate Double-fetch Bugs

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit
- Compiler can introduce double-fetch bugs

# Eliminate Double-fetch Bugs

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit
- Compiler can introduce double-fetch bugs
- Even frameworks to exploit such race conditions [Wei+16]

# Eliminate Double-fetch Bugs

- Idea: Ensure that both accesses are atomic...

# Eliminate Double-fetch Bugs

- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective

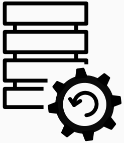
# Eliminate Double-fetch Bugs

- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective
- We have such a mechanism on modern Intel CPUs

# Eliminate Double-fetch Bugs

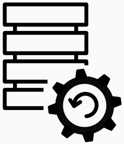
- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective
- We have such a mechanism on modern Intel CPUs
- Intel TSX provides exactly this functionality in hardware

# Transactional Memory



- Intel TSX is an implementation of hardware transactional memory

# Transactional Memory



- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic

# Transactional Memory

- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction

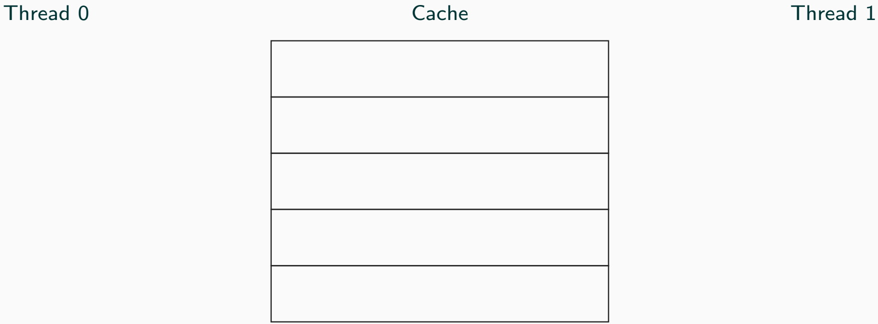
# Transactional Memory

- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction
- If something conflicts, transaction is rolled back

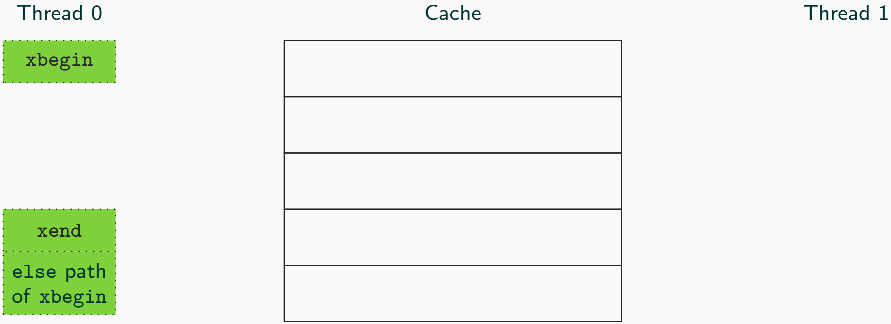
# Transactional Memory

- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction
- If something conflicts, transaction is rolled back
- Implemented using the cache

# Transactional Memory



# Transactional Memory



Thread 0

Cache

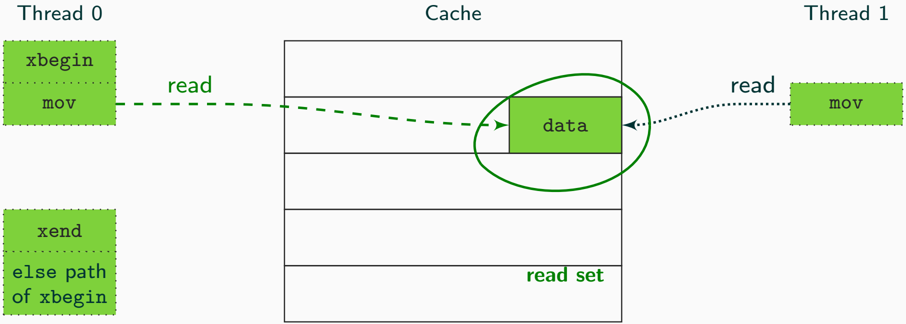
Thread 1

`xbegin`

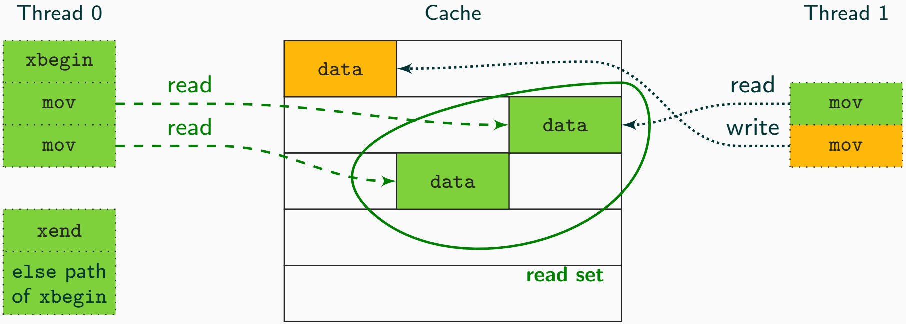
`xend`

`else path of xbegin`

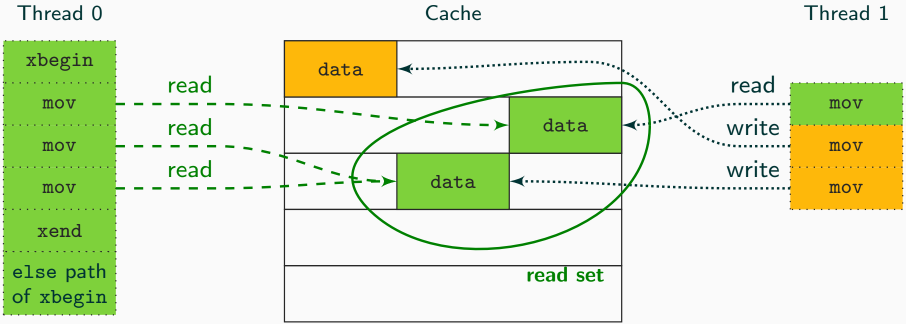
# Transactional Memory



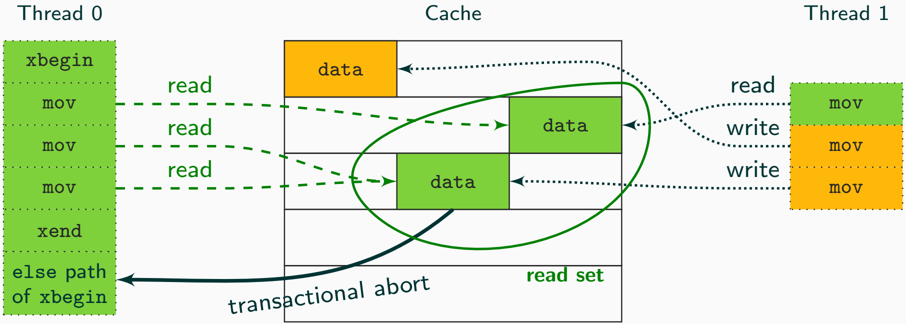
# Transactional Memory



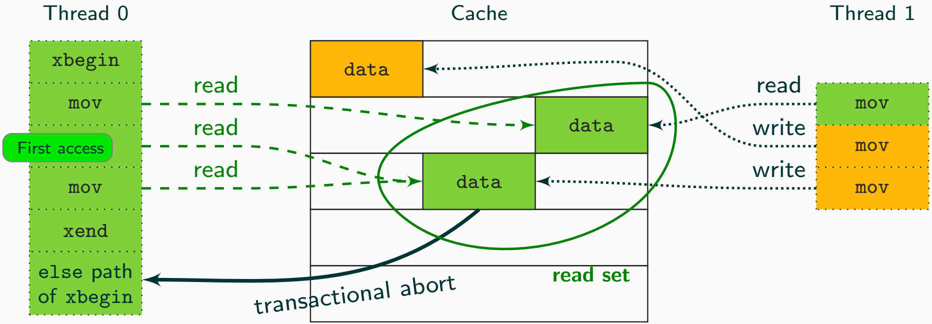
# Transactional Memory



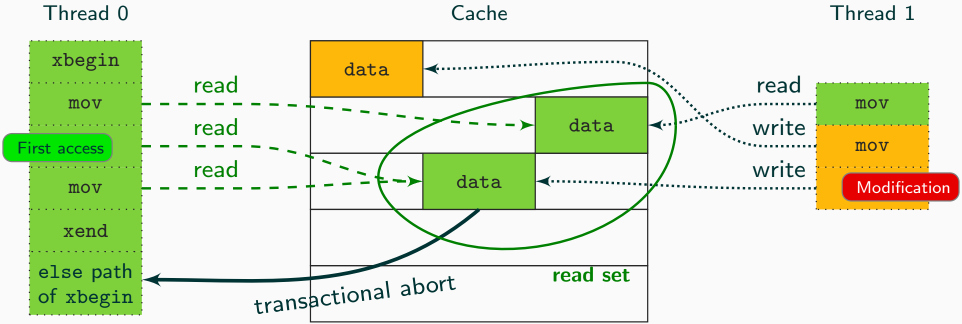
# Transactional Memory



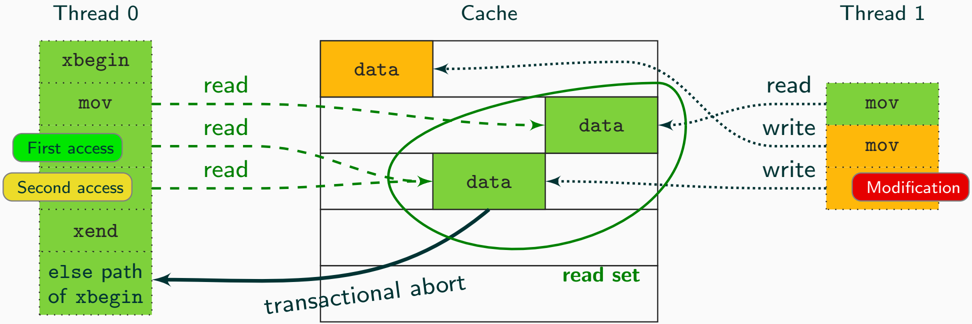
# Transactional Memory



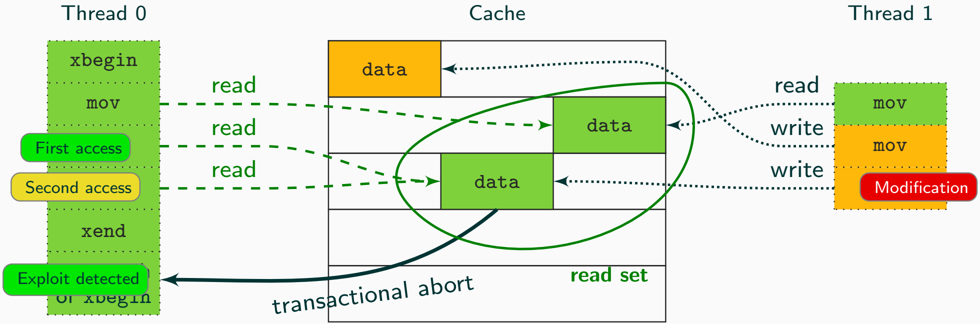
# Transactional Memory



# Transactional Memory



# Transactional Memory



Thread 0

Cache

Thread 1

xbegin

mov

First access

Second access

xend

Exploit detected

of xbegin

read

read

read

data

data

data

read set

transactional abort

read

write

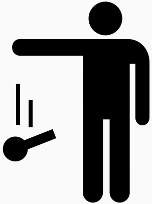
write

mov

mov

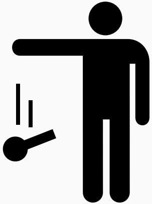
Modification

# DropIt: TSX against Double-fetch Bugs



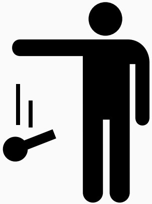
- We created DropIt, a tiny library

# DropIt: TSX against Double-fetch Bugs



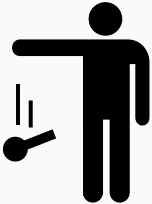
- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX

# DropIt: TSX against Double-fetch Bugs



- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX
- Open source: `https://github.com/IAIK/libdropit`

# DropIt: TSX against Double-fetch Bugs



- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX
- Open source: `https://github.com/IAIK/libdropit`
- Easy to use, 3 additional lines of code are sufficient

# A Double Fetch

```
char buffer[8];
size_t len;
// init DropIt
doublefetch_t config = doublefetch_init(10);
// start DropIt
doublefetch_start(config);
// first access to string
len = strlen(string);
if(len < 8) {
    // second access to string
    strcpy(buffer, string);
} else {
    printf("String too long!\n");
}
// end of critical section
doublefetch_end(config, {
  printf("Exploit detected!\n"); exit(-1);
});
```

# A Double Fetch

Retry maximum 10 times

```
char buffer[8];
size_t len;
// init DropIt
doublefetch_t config = doublefetch_init(10);
// start DropIt
doublefetch_start(config);
// first access to string
len = strlen(string);
if(len < 8) {
    // second access to string
    strcpy(buffer, string);
} else {
    printf("String too long!\n");
}
// end of critical section
doublefetch_end(config, {
  printf("Exploit detected!\n"); exit(-1);
});
```

# A Double Fetch

Start protection

Retry maximum 10 times

```
char buffer[8];
size_t len;
// init DropIt
doublefetch_t config = doublefetch_init(10);
// start DropIt
doublefetch_start(config);
// first access to string
len = strlen(string);
if(len < 8) {
    // second access to string
    strcpy(buffer, string);
} else {
    printf("String too long!\n");
}
// end of critical section
doublefetch_end(config, {
  printf("Exploit detected!\n"); exit(-1);
});
```

# A Double Fetch

Retry maximum 10 times

Start protection

End protection

```
char buffer[8];
size_t len;
// init DropIt
doublefetch_t config = doublefetch_init(10);
// start DropIt
doublefetch_start(config);
// first access to string
len = strlen(string);
if(len < 8) {
    // second access to string
    strcpy(buffer, string);
} else {
    printf("String too long!\n");
}
// end of critical section
doublefetch_end(config, {
  printf("Exploit detected!\n"); exit(-1);
});
```

# A Double Fetch

```
char buffer[8];
size_t len;
// init DropIt
doublefetch_t config = doublefetch_init(10);
// start DropIt
doublefetch_start(config);
// first access to string
len = strlen(string);
if(len < 8) {
    // second access to string
    strcpy(buffer, string);
} else {
    printf("String too long!\n");
}
// end of critical section
doublefetch_end(config, {
  printf("Exploit detected!\n"); exit(-1);
});
```

Retry maximum 10 times

Start protection

Everything in between is atomic

End protection

# DropIt: TSX against Double-fetch Bugs

- Efficient and easy solution to a complex problem

# DropIt: TSX against Double-fetch Bugs

- Efficient and easy solution to a complex problem
- Faster than traditional locking ($\approx 18\,\%$)

# DropIt: TSX against Double-fetch Bugs

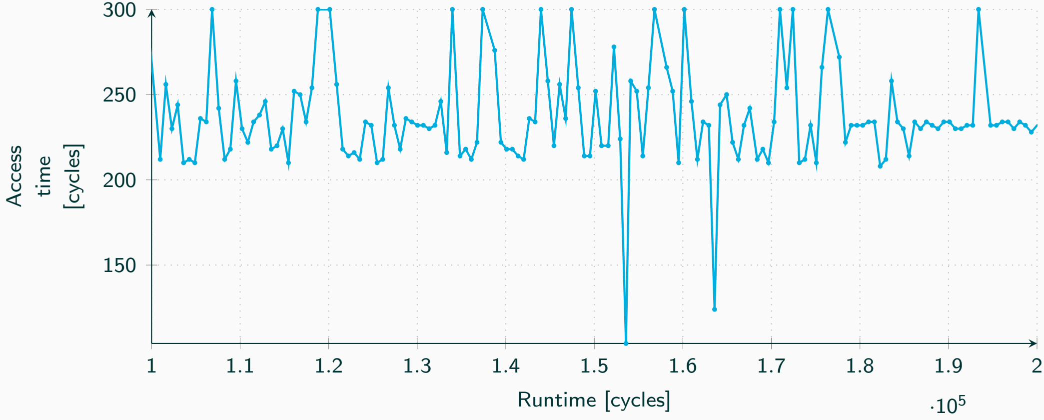
- Efficient and easy solution to a complex problem
- Faster than traditional locking ($\approx 18\,\%$)
- Unfortunately limited to new Intel CPUs

# DropIt: TSX against Double-fetch Bugs

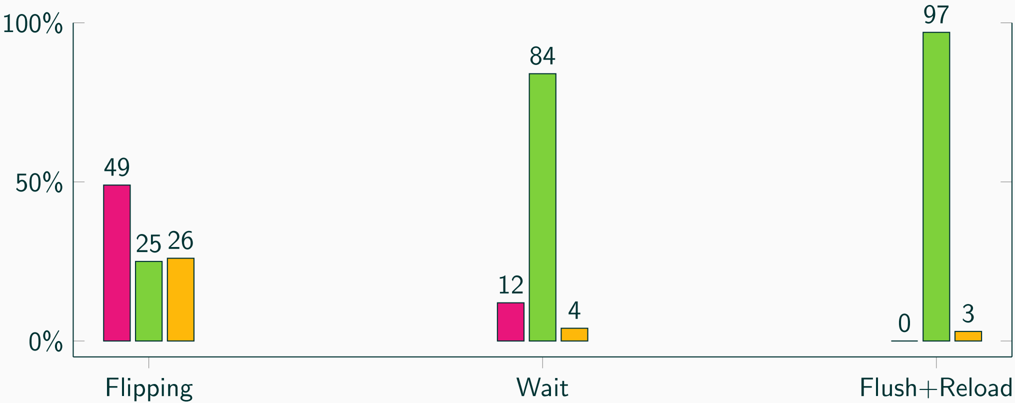
- Efficient and easy solution to a complex problem
- Faster than traditional locking (≈ 18 %)
- Unfortunately limited to new Intel CPUs
- Can be automatically applied in some environments (e.g., SGX)

# A Real-World Double-Fetch Bug

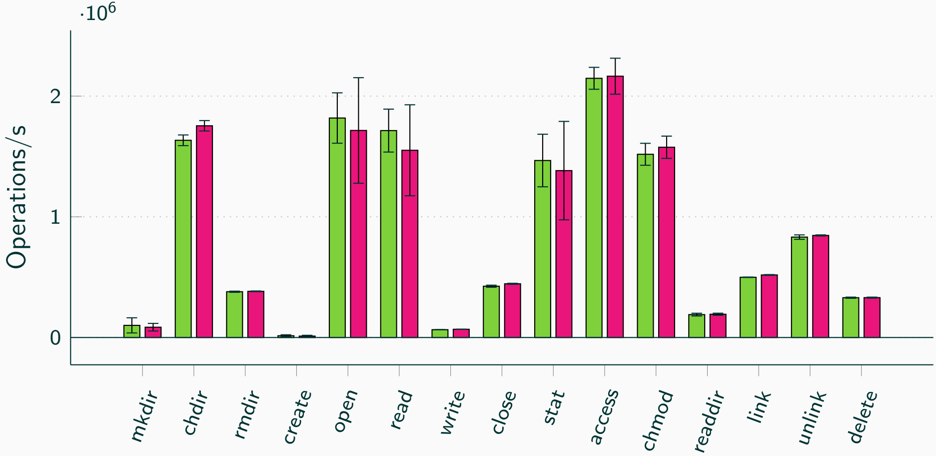
# $\mathcal{P}1$: CVE-2016-6516



# $\mathcal{P}$2: CVE-2016-6516



# $\mathcal{P}3$: CVE-2016-6516



# Conclusion

# Conclusion

- Fuzzing augmented with cache attacks automatically finds double-fetch bugs

# Conclusion

- Fuzzing augmented with cache attacks automatically finds double-fetch bugs
- Can be added to any fuzzer

# Conclusion

- Fuzzing augmented with cache attacks automatically finds double-fetch bugs
- Can be added to any fuzzer
- Cache as exploitation trigger outperforms state of the art

# Conclusion

- Fuzzing augmented with cache attacks automatically finds double-fetch bugs
- Can be added to any fuzzer
- Cache as exploitation trigger outperforms state of the art
- Hardware transactional memory can prevent double-fetch bug exploitation

**Thank you!**

# Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features

---

**Michael Schwarz**, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, Stefan Mangard

June 7, 2018

M. Jurczyk, G. Coldwind, et al. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In: ESORICS'16. 2016.

P. Wang and J. Krinke. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In: USENIX Security Symposium. 2017.