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
string
A Double Fetch

Thread 1

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

Thread 2

```c
string
```

/ path / file \0payload \0

length
A Double Fetch

Thread 1

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

// <switch to kernel>

Thread 2
Thread 1

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

// <switch to kernel>

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

Thread 2
string

```
/path/file\0payload
```

Thread 1

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

// <switch to kernel>

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

Thread 2

```
string[10] = 'X';
```
A Double Fetch

Thread 1

```c
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

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

Michael Schwarz (@misc0110) et al.
- Not all double fetches are exploitable
Double Fetches

- Not all double fetches are exploitable
- Changing data after sanity check allows exploitation
• 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
• 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
• A dynamic approach based on fuzzing and cache attacks
• A dynamic approach based on fuzzing and cache attacks
• Basic idea: memory access can be observed in the cache
- 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
• 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
(Syscall) Fuzzer

Report general bug
DECAF

(Syscall) Fuzzer

Report general bug

DECAF

Detect double fetches ($P_1$)

Double fetch candidates

Report double-fetch bug

Fix double-fetch bug ($P_3$), e.g., DropIt

Michael Schwarz (@misc0110) et al.
DECAF

(Syscall) Fuzzer

DECAF

Detect double fetches (P1)

Report general bug

Michael Schwarz (@misc0110) et al.
(Syscall) Fuzzer

DECAF

Detect double fetches ($P1$)

Double fetch candidates

Report general bug
DECAF

(Syscall) Fuzzer

DECAF

Exploit double fetch (P2)

Detect double fetches (P1)

Double fetch candidates

Report general bug

P1

P2

Michael Schwarz (@misc0110) et al.
DECAF

(Syscall) Fuzzer

DECAF

Exploit double fetch ($P_2$)

Detect double fetches ($P_1$)

Double fetch candidates

Report general bug

Report double-fetch bug

Michael Schwarz (@misc0110) et al.
DECAF

(Syscall) Fuzzer

DECAF

Exploit double fetch ($\mathcal{P}_2$)

Detect double fetches ($\mathcal{P}_1$)

Double fetch candidates

Report general bug

Fix double-fetch bug ($\mathcal{P}_3$), e.g., DropIt

Report double-fetch bug

Michael Schwarz (@misc0110) et al.
P1: Detection
• Shared memory make cache observation easy
• Shared memory make cache observation easy
• Constantly flush value from cache using clflush instruction
- Shared memory make cache observation easy
- Constantly flush value from cache using clflush instruction
- Schedule victim
- Shared memory make cache observation easy
- Constantly flush value from cache using clflush instruction
- Schedule victim
- Measure access time to value
• 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

Runtime [cycles]

Access time [cycles]
Data was accessed
Double-fetch Detection

![Graph showing access time vs. runtime in cycles]

Michael Schwarz (@misc0110) et al.
Double-fetch Detection

Runtime [cycles]

Access time [cycles]

First access

Michael Schwarz (@misc0110) et al.
Double-fetch Detection

![Graph showing access time vs. runtime with first and second access highlighted.](image-url)
Cache attacks *dynamically* detect double fetches.
- Cache attacks *dynamically* detect double fetches
- The further apart the two fetches, the higher the probability
Cache attacks dynamically detect double fetches.
The further apart the two fetches, the higher the probability.
Minimum time between accesses is approximately 600 cycles.
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 $\geq 3000$ cycles, detection rate is close to 100%.
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 \( \geq 3000 \) 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
Not limited to syscalls and not limited to operating system
Oblivious to programming language and programming constructs
• 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

Michael Schwarz (@misc0110) et al.
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

Michael Schwarz (@misc0110) et al.
P2: Exploitation
Double-fetch Bug Exploitation

- Double fetches are not necessarily exploitable
• Double fetches are **not necessarily exploitable**
• Double-fetch bug if exploitable
• 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
• 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
• Use the cache side channel as trigger
- Use the cache side channel as trigger
- The first memory access acts as 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

![Graph showing access time versus runtime with markers for first access and modify value.]

Michael Schwarz (@misc0110) et al.
Cache-based Trigger

Access time [cycles]

Runtime [cycles]

First access
Modify value
Second access with modified value

Michael Schwarz (@misc0110) et al.
Cache-based Trigger

Michael Schwarz (@misc0110) et al.
P3: Prevention
• Double fetches are not bad by itself
Double fetches are not bad by itself

Sometimes they are necessary
• 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...

Michael Schwarz (@misc0110) et al.
• 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
• Intel TSX is an implementation of hardware transactional memory
• Intel TSX is an implementation of hardware transactional memory
• Ensures that multiple reads and writes are atomic
• Intel TSX is an implementation of hardware transactional memory
• Ensures that multiple reads and writes are atomic
• Operations are wrapped in a transaction
- 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**
• 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

Thread 0

Cache

Thread 1
Transactional Memory

Thread 0

xbegin

xend
else path of xbegin

Cache

Thread 1
Transactional Memory

Thread 0

\begin{align*}
\text{xbegin} \\
\text{mov}
\end{align*}

read

\begin{align*}
\text{xend} \\
\text{else path of xbegin}
\end{align*}

Cache

\begin{align*}
data \\
\text{read set}
\end{align*}

read

Thread 1

\begin{align*}
\text{mov}
\end{align*}
Transactional Memory

Thread 0
- \texttt{xbegin}
- \texttt{mov}
- \texttt{mov}
- \texttt{xend}
- else path of \texttt{xbegin}

Thread 1
- \texttt{mov}
- \texttt{mov}

Cache
- data
- data
- read set

\texttt{mov} \quad \texttt{mov} \quad \texttt{mov}

Michael Schwarz (@misc0110) et al.
Transactional Memory

Thread 0
- xbegin
- mov
- mov
- mov
- xend
- else path of xbegin

Thread 1
- read
- write
- read
- write
- read

Cache
- data
- data
- data

read set

Michael Schwarz (@misc0110) et al.
Thread 0

- xbegin
- mov
- First access
- mov
- xend
- else path of xbegin

Thread 1

- read
- read
- write
- mov
- mov
- mov

Cache

- data
- read set

Transactional abort

Michael Schwarz (@misc0110) et al.
Transactional Memory

Thread 0
- `xbegin`
- `mov`
- `First access`
- `mov`
- `mov`
- `xend`
- `else path of xbegin`---

Thread 1
- `mov`
- `read`
- `write`
- `mov`
- `read`
- `write`
- `Modification`

Cache
- `data`
- `read set`
- `transactional abort`
Transactional Memory

Thread 0

- `xbegin`
- `mov`

First access

Second access

- `xend`
- else path of `xbegin`

Thread 1

Cache

- `data`
- `read`
- `mov`

- `mov`
- `write`

Modification

Transaction abort

Read set

First access

Second access

Modification

Write set
Transaction Memory

Thread 0

\texttt{xbegin}
\texttt{mov}

First access

Second access

\texttt{xend}

Exploit detected or \texttt{xbegin}

\texttt{read}

\texttt{read}

\texttt{read}

\texttt{read}

\texttt{read}

Cache

\texttt{data}

\texttt{data}

\texttt{data}

\texttt{data}

\texttt{read set}

\texttt{transactional abort}

Thread 1

\texttt{mov}

\texttt{mov}

\texttt{read}

\texttt{write}

\texttt{write}

Modification

Michael Schwarz (@misc0110) et al.
We created DropIt, a tiny library.
• We created DropIt, a tiny library
• It prevents exploitation of double fetches using TSX
We created DropIt, a tiny library.

It prevents exploitation of double fetches using TSX.

Open source: 🌐 https://github.com/IAIK/libdropit
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.
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);
});
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);
});
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);
});
```c
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);
});
```
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);
});
DropIt: TSX against Double-fetch Bugs

- Efficient and easy solution to a complex problem
Efficient and easy solution to a complex problem
Faster than traditional locking ($\approx 18\%$)
• Efficient and easy solution to a complex problem
• Faster than traditional locking (≈ 18%)
• Unfortunately limited to new Intel CPUs
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
Figure 1: The graph shows the relationship between runtime and access time. The x-axis represents Runtime [cycles] ranging from 1 to 2, and the y-axis represents Access time [cycles] ranging from 0 to 300. The data points indicate fluctuations in access time as the runtime increases.
Operations/s

mkdir   chdir   rmdir   create   open   read   write   close   stat   access   chmod   readdir   link   unlink   delete

\textbf{P3: CVE-2016-6516}

Michael Schwarz (@misc0110) et al.
Conclusion
• Fuzzing augmented with cache attacks automatically finds double-fetch bugs
Fuzzing augmented with cache attacks automatically finds double-fetch bugs
Can be added to any fuzzer
• 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
• 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
