SpecFuzz
Bringing Spectre-type vulnerabilities to the surface

Oleksii Oleksenko
Motivation
BOO!
Not scary at all? Cute? Cuddly?
Not scary at all? Cute? Cuddly?
Buffer overflow

\[ y = \text{array}[x]; \]

x can be larger than the array size
if (x >= 0 && x < size) {
    y = array[x];
}
I SHALL BYPASS!
\[
x = 1
\]

// size = 10
if (x >= 0 && x < size) { True
    y = array[x];
}


\[ x = 3 \]

// size = 10
if (x >= 0 && x < size) { True
    y = array[x];
}

x = 2

// size = 10
if (x >= 0 && x < size) {
    y = array[x];
}

x = 1 Gazillion

// size = 10
if (x >= 0 && x < size) {
    y = array[x];
}
What will happen to us?
Hm... Let me see your history
You passed once, then again, and again...
You will pass!
Bounds check bypass

if (x >= 0 && x < size) {
    y = array[x];
}

Predict true
Execute speculatively
Bounds check bypass

if (x >= 0 && x < size) {
    y = array[x];
}

Speculative execution:
- Not visible to software
- Leaves detectable traces in hardware
Isn't it a CPU bug?
<table>
<thead>
<tr>
<th>CPU Model and Stepping</th>
<th>V1, Spectre</th>
<th>V2, Spectre</th>
<th>V3, Meltdown</th>
<th>V3a</th>
<th>V4</th>
<th>L1TF, Foreshadow</th>
<th>MFBDS, RIDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel64 Family 6 Model 142 Stepping 11</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 142 Stepping 12</td>
<td>Software</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 11</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Software</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>MCU + Software</td>
<td>MCU + Software</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 12</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 13</td>
<td>Software</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>CPU Model and Stepping</td>
<td>V1, Spectre</td>
<td>V2, Spectre</td>
<td>V3, Meltdown</td>
<td>V3a</td>
<td>V4</td>
<td>L1TF, Foreshadow</td>
<td>MFBDS, RIDL</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
<td>------------</td>
<td>--------------</td>
<td>-----</td>
<td>----</td>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 142 Stepping 11</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 142 Stepping 12</td>
<td>Software</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 11</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Software</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>MCU + Software</td>
<td>MCU + Software</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 12</td>
<td>Software</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>MCU + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
<tr>
<td>Intel64 Family 6 Model 158 Stepping 13</td>
<td>Software</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>MCU</td>
<td>Hardware + Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
</tbody>
</table>
SPECTRE V1 IS A FLAW IN OUR PRODUCTS

BUT THAT'S NONE OF OUR BUSINESS

PRETTY MUCH ANY CPU VENDOR

BUT THAT'S NONE OF OUR BUSINESS
What can I do?
SERIALIZE

ALL THE THINGS!
if (x >= 0 && x < size) {
    __mm_lfence(); // stops speculation
    y = array[x];
}
if (x >= 0 && x < size) {
    __mm_lfence();  // stops speculation
    y = array[x];
}
ADD
DATA
DEPENDENCY TO
ALL THE THINGS!
if (x < size) {
    x = (x < size) ? x : 0;
    y = array[x];
}
if (x < size) {
  x = (x < size) ? x : 0;
y = array[x];
}
We need more precision!
How do we find Spectre?
How do we find other bugs?
Fuzzing!
x = generate_randomized_int();

if (x >= 0 && x < size) {
    y = array[x];
}
x = generate_randomized_int();

if (x >= 0 && x < size) {
    __asan_check_if_valid(array + x);
    y = array[x];
}

x = generate_randomized_int();

if (x >= 0 && x < size) {
    __asan_check_if_valid(array + x);
    y = array[x];
}

Always valid!
How do we make speculative execution visible?
Let's simulate it!
ENTER

checkpoint

A

A_{\text{sim}}

B

C

D

rollback

simulation? YES

sim

FALSE

TRUE

NO

EXIT
A
 checkpoint

A_{sim}

B
 TRUE

C
 FALSE

D
 simulation?

rollback

YES

EXIT

NO
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}

<victim_function>:
    CMP %rdi, size
    .if: JL .else
    MOV array(%rdi), %eax
    AND %al, result
    .else: RET
```c
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}
```
```c
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}
```

```assembly
<victim_function>:
    CMP %rdi, size
    .if:    JL .else
    MOV array(%rdi), %eax
    AND %al, result
    .else:  RET
    .if:    JL .else
    CALL specfuzz_chkp
    MOV (%rdi), %eax
    AND %al, result
    .else:  RET  
```

Checkpoint + mispredict
```c
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}
```

```
MOV (%rdi), %eax
AND %al, result
else:
    RET
```

Checkpoint + mispredict
void victim_function(size_t x) {
  if (x < size) {
    result &= array[x];
  }
}

Checkpoint + mispredict

Rollback
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}

Checkpoint + mispredict

Rollback if counter > 250
```c
void victim_function(size_t x) {
    if (x < size) {
        result &= array[x];
    }
}

<victim_function>:
    CMP %rdi, size
.if:    JL .else
        MOV array(%rdi), %eax
        AND %al, result
.else:  RET

 Skip:
    SUB $0x2, instruction_counter
    LEA array(%rdi), %rdi
    CALL __asan_load1
    MOV (%rdi), %eax
    AND %al, result
    CALL specfuzz_maybe_rlbk
.else:  RET
```

- **Checkpoint + mispredict**: `CALL specfuzz_chkp` after a branch misprediction.
- **Bounds check**: `CALL __asan_load1` to check for array bounds.
- **Rollback if counter > 250**: `CALL specfuzz_maybe_rlbk` to rollback if the instruction counter exceeds 250.
Demo

Fuzzing OpenSSL
/fuzz/corpora/server -l openssl.log -- $fuzz/server FILE 2>&1 |
 analyzer.py collect -r openssl.log -o analyzer.json -b ./fuzz/server
Now what?
Whitelist patching

Instrument all branches except:

- Covered
- No vulnerabilities detected
Speedup

Higher is better
Speedup

Higher is better
Want more?
See our paper!

SpecFuzz
Bringing Spectre-type vulnerabilities to the surface

Oleksii Oleksenko†, Bohdan Trach†, Mark Silberstein‡, and Christof Fetzer†
†TU Dresden, ‡Technion

Abstract
SpecFuzz is the first tool that enables dynamic testing for speculative execution vulnerabilities (e.g., Spectre). The key is a novel concept of speculation exposure: The program is instrumented to simulate speculative execution in software by forcefully executing the code paths that could be triggered due to mispredictions, thereby making the speculative memory accesses visible to integrity checkers (e.g., AddressSanitizer). Combined with the conventional fuzzing techniques, speculation exposure enables more precise identification of potential Intel [28]. Therefore, the burden of protecting programs lies entirely on software developers [40].

Unfortunately, existing software mitigation tools suffer either from high performance penalty or from low precision. Conservative techniques [3, 11, 21, 51] pessimistically harden every speculatable instruction such as conditional branches, to either prevent the speculation or make it provably benign. The techniques, however, often result in a high performance overhead, significantly slowing down applications [44].

Another defense strategy is to use static analysis tools [18,
https://github.com/tudinfse/SpecFuzz
Warning!
Academic Code

https://github.com/tudinfse/SpecFuzz
SpecFuzz

Questions?

@oleksii_o

oleksii.oleksenko@tu-dresden.de
Backup
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>

extern size_t array1_size, array2_size;
extern uint8_t temp, array2[], array1[];

void victim_function(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] + 512];
    }
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("USAGE: %s <index> \n", argv[0]);
        exit(1);
    }

    int index = atoi(argv[1]);
    victim_function(index);
    printf("r = %d\n", temp);
    return 0;
}
(a) Control Flow Graph

(b) A's Simulation Tree