Countermeasures Against Branch Target Buffer Attacks

Giovanni Agosta, Luca Breveglieri
Politecnico di Milano

Gerardo Pelosi
Università degli Studi di Bergamo

Israel Koren
University of Massachusetts at Amherst
Outline

- Microarchitecture Overview
- BTB Attack
- State of the art countermeasures
- Proposed countermeasures:
  - Predicated Execution
  - Indirect Jump Conversion
- Performance Evaluation
- Concluding Remarks
Microarchitecture Overview

Branch Prediction

- Dynamic prediction of a branch outcome is based on a two-bit saturating counter that is an entry of a Branch Prediction Table (BPT)
- The BHR is a shift register that keeps the history of most recent branch outcomes
- BPT is indexed by a portion of the branch address or a combination of the branch address with a branch history register (BHR)

Branch Target Buffer (BTB) is a cache structure indexed by the low order part of the branch address; the cache data is the last target address of that branch
Two-bit predictors are used to improve performance over one-bit predictors (MR=2/k+1 for 1-bit predictors)
**BTB Attack - Basic Principle**

- **Simultaneous Multithreaded Processors (SMPs)** execute two threads at the same time.
  - One physical CPU but two logical CPUs: in the same cycle, instructions from the two threads are executed on different execution units in the CPU.

- **HW information leakage is feasible** (exploited by Acıçmez, Koç & Seifert) due to the sharing of the branch target buffer (BTB) by all threads.
  - A simultaneous spy-thread can be launched to discover indirect information about execution flow of another thread.
  - The collected log data can be used to make educated guesses of bits of an encryption key.
BTB Attack on RSA

The core of the RSA algorithm includes a loop that handles modular squaring and multiplication:

- The former (squaring) is always executed.
- The latter (multiplication) is executed only if the key bit is 1.

Attack Scenario:

- A crypto process performs an RSA encryption operation.
- An attacking spy process executes a sufficient number of branches to replace the BTB block used by the crypto process.
- The crypto-process is forced to have mispredicted branches when it is about to compute a multiplication.
- The spy-process measures the time needed to perform its own branches and is able to determine whether a branch was taken or not in the crypto process by observing the mispredictions occurring during its own code execution.
Countermeasures: state of the art

- **Coron’s Method**

  \[ \text{if (a) \{ b = c+d \}} \]

  - Limitation: unsecure w.r.t. attacks that exploit knowledge of accessed data memory addresses

- **Program Counter-Secure code [Molnar et. al]**
  - Remove all conditional branches from a program so that all execution traces have the same sequence of PC values
  - Limitation: some conditional statements can be driven at runtime only (e.g. input values)
  - Experiments reported by the authors show performance slowdown of up to 5x and an increased stack size of up to 2x

\[
\text{tmp}[1] = b+c \\
\text{tmp}[0] = b \\
b = \text{tmp}[a]
\]
Countermeasure - Predicated Execution

Sensitive branches are implemented as instructions belonging to a single control flow

```c
if (a) { b = c+d }
```

```c
cmpl r1, r2, 0
add r3, r4, r5
select r2, r3, r1

// if (r2 == 0) then { r1 = 1 }
// else { r1 = 0 }

// r3 = r4 + r5
// if (r1 != 0) then { r2 = r3 }
```
Countermeasure - Indirect Jump

- Replace all conditional branches in sensitive code by equivalent indirect jumps
- A specific BTB entry (fixed position) will always be changed by the attack process independent of program logic

```assembly
// r1 is 0 or 1 based on the condition expression
bz r1, label // branch to label if r1 is zero
<then statement >
jmp end
label: <else statement >
end:
```

// [r3] == mem. addr of <then block >
// [r3]+1 == mem. addr of <else block >
add r2, r3, r1 // r2 ← [r3] + [r1]
load r4, 0(r2) // r4← [0+[r2]]
jmpl r4 // PC ← [r4]

Spy-process will cause the branch to be always mispredicted, but will also find its own branches to be always mispredicted - the attacked process also changes the specific BTB entry for each execution
Indirect Jump Conversion

- Applicable to high level source codes by replacing *if-then-else* statements with an ad-hoc macro (simple compiler pass with minimal overhead)
- Directly applicable to binary code when basic blocks position in memory is known (to secure closed source cryptographic SW)
- Easily implementable at link-time or in dynamic-optimizers
- Each branch is still executed on different sets of PC values but is effective against BTB attacks with negligible performance impact w.r.t. PC-secure method
### Performance Evaluation

<table>
<thead>
<tr>
<th>Method</th>
<th>Branch penalty</th>
<th>Footprint penalty</th>
<th>Data Ref. penalty</th>
<th>Time [clk]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Code</td>
<td>1.00</td>
<td>1.0</td>
<td>0</td>
<td>59,698</td>
</tr>
<tr>
<td>Coron</td>
<td>1.71</td>
<td>0.8</td>
<td>2</td>
<td>58,756</td>
</tr>
<tr>
<td>Predicated conditional jump</td>
<td>4.79</td>
<td>1.2</td>
<td>4</td>
<td>58,967</td>
</tr>
<tr>
<td>Indirect Jump</td>
<td>4.83</td>
<td>2.0</td>
<td>3</td>
<td>61,846</td>
</tr>
</tbody>
</table>

Branch, Footprint and Data ref. penalties refer to a single branch.

Execution time is given in clock cycles for 1024-RSA kernel loop.
The proposed countermeasures have a minimal impact on the memory usage profile
Concluding Remarks

- We surveyed several SW countermeasures against BTB side-channel attacks.
- Molnar’s method gives the maximum security but has a high overhead (5x slowdown).
- The Indirect Jump method is both effective and has low overhead (less than 1.05x slowdown) and can be applied selectively, automatically, and without special HW support.