Countermeasures Against Branch Target Buffer Attacks

Giovanni Agosta, Luca Breveglieri

Politecnico di Milano

Gerardo Pelosi

Università degli Studi di Bergamo

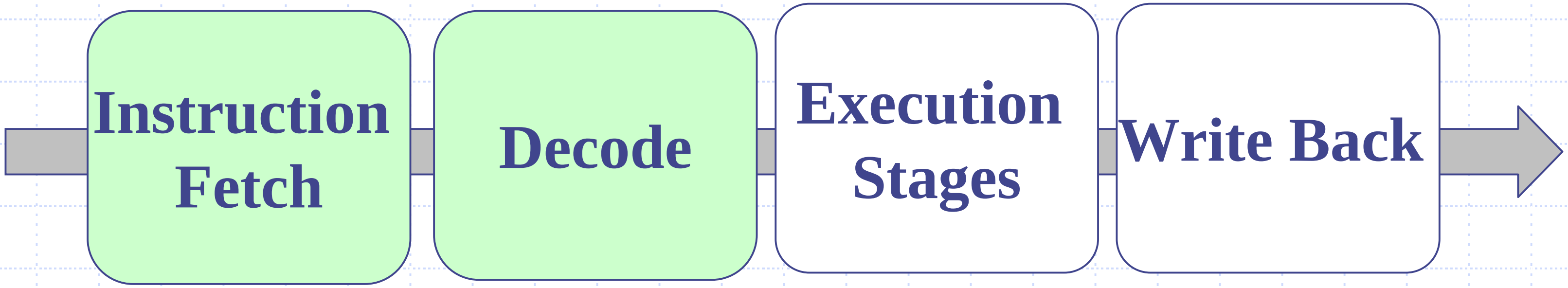
Israel Koren

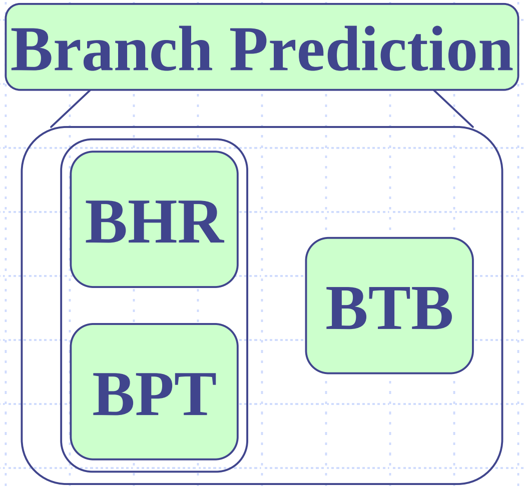
University of Massachusetts at Amherst

Outline

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

Microarchitecture Overview

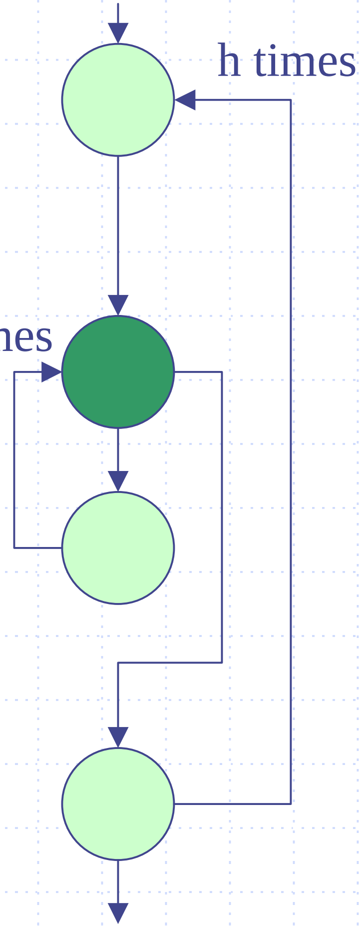




- Dynamic prediction of a branch outcome is based on a two-bit saturating counter that is an entry of a **B**ranch **P**rediction **T**able (**BPT**)
- The **BHR** is a shift register that keeps the history of most recent branch outcomes
- **BPT** is indexed by a portion of the branch address or a combination 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

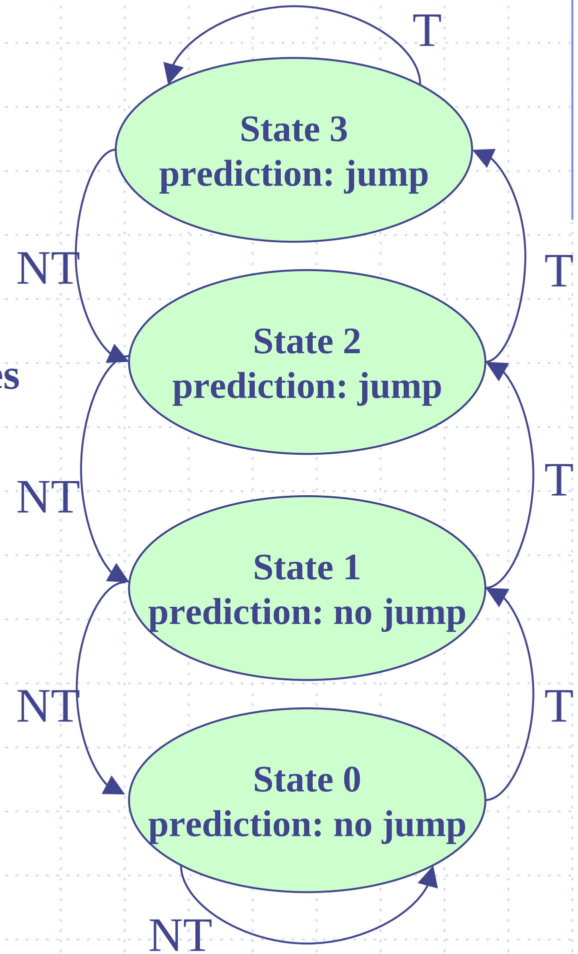
FSM for Dynamic Prediction



Prediction	Outcome (*Taken / Not Taken*)
N0	NT
...	...
N0	NT
N0	T
N1	NT
N0	NT

k times (rows N0 / NT through N0 / NT)

Misprediction rate $= \dfrac{1}{k+1}$



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 *Acıiçmez, 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*:

  if (a) { b = c+d } → *tmp[1] = b+c*
  tmp[0] = b
  b = tmp[a]

  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

Countermeasure - Predicated Execution

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

if (a) { b = c+d }

```
cmpi r1, r2, 0        // if (r2 ==  0) then { r1 = 1 }
                      //                else { r1 = 0 }
                      //
add r3, r4, r5        // r3 = r4 + r5
select r2, r3, r1     // 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

```
// 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

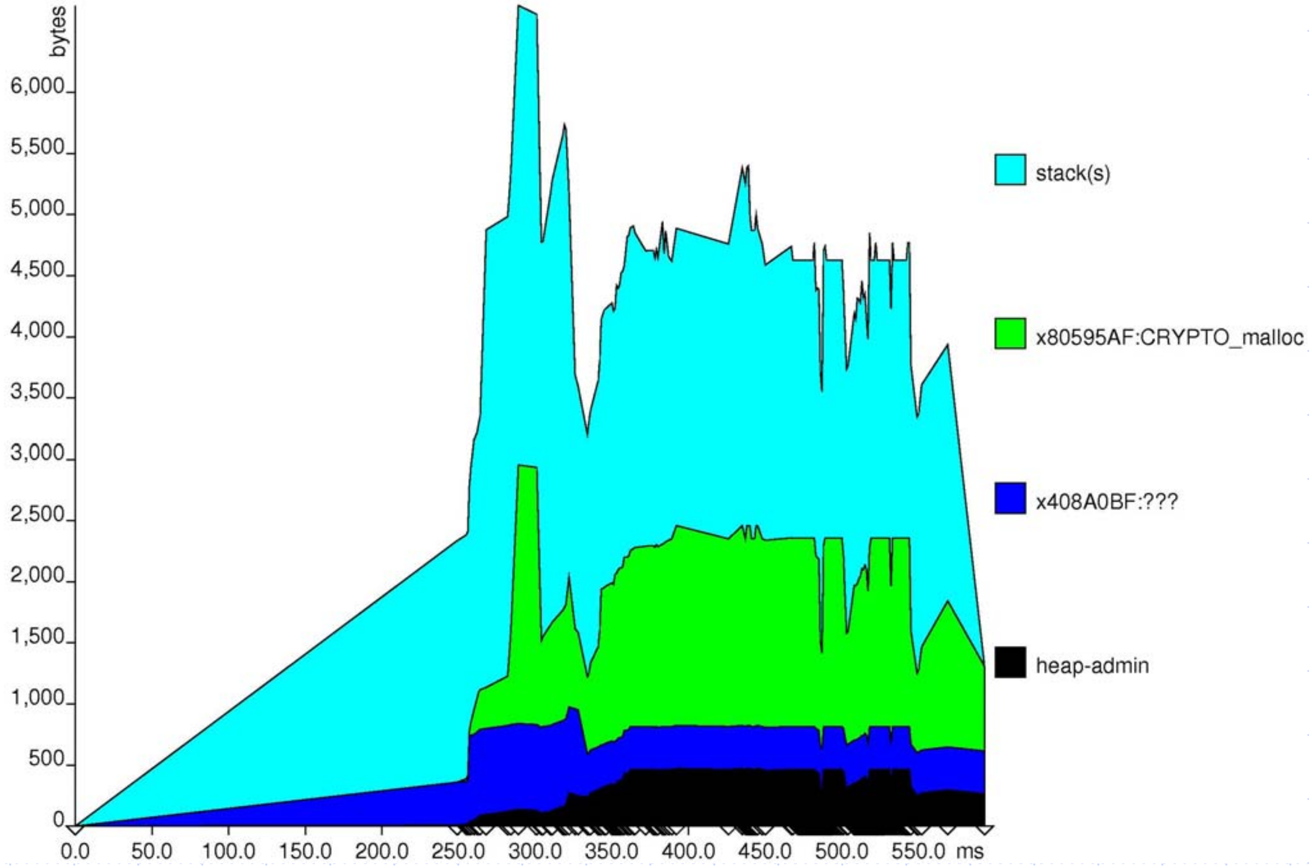
- Applicable to high level source codes by replacing *if-then-else* statements with an ad-hoc macro (simple compiler pass with minimal overhead)
- Directly applicable to binary code when basic blocks position in memory is known (to secure closed source cryptographic SW)
- Easily implementable at link-time 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

Method	Branch penalty	Footprint penalty	Data Ref. penalty	Time [clk] 1024-RSA S&M
Original Code	1.00	1.0	0	59,698
Coron	1.71	0.8	2	58,756
Predicated conditional	**4.79**	**1.2**	**4**	**58,967**
Indirect Jump	**4.83**	**2.0**	**3**	**61,846**

- Branch, Footprint and Data ref. penalties refer to a single branch
- Execution time is given in clock cycles for 1024-RSA kernel loop

Memory usage in RSA S&M



- 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