Predicting the Branch Predictors: inception to secure predictors

Sarani Bhattacharya

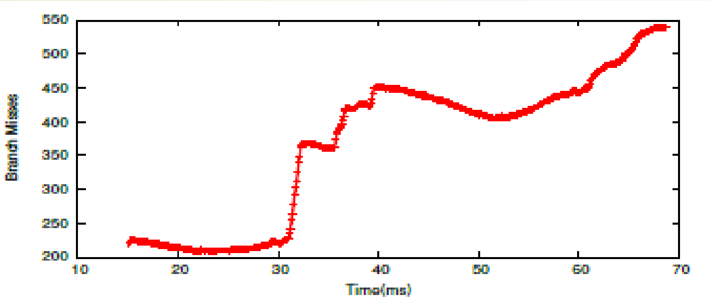
COSIC, KU Leuven

Branch Prediction

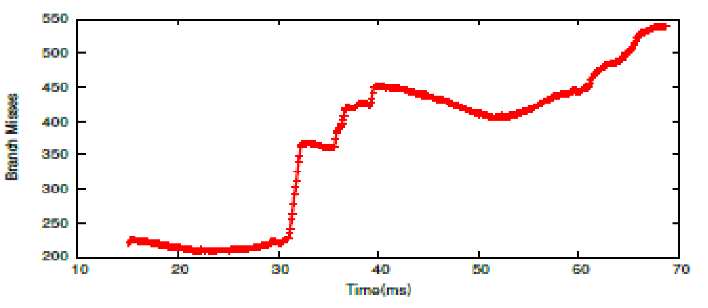
Motivation of the work

- Computer Architecture has evolved over the decade with performance improvisation being its sole motivation and objective.
- In this work, we start with the security evaluation of one of the most important architectural component- the branch predictors.
- It is also a difficult task to guess which particular design has been implemented in hardware- this requires basic reverse engineering.
- There exists no security guidelines to sensitive cryptographic applications executing in multi-tenant or cloud environment. There still exists legacy codes like RELIC and texts books which suggest such implementation to be efficient, though being highly vulnerable to micro-architectural attacks.

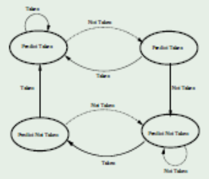
Observation 1



Abrupt increase in branch miss observed by unprivileged user due to exponentiations from privileged process



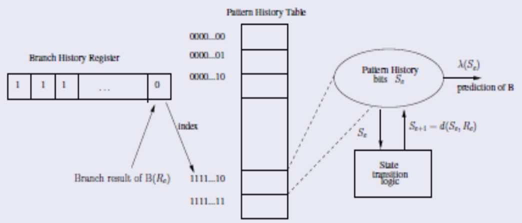
Abrupt increase in branch miss observed by unprivileged user due to exponentiations from privileged process

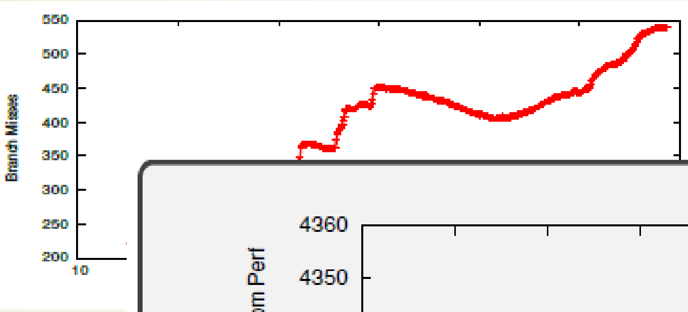


Dynamic 2-bit predictor State Machine

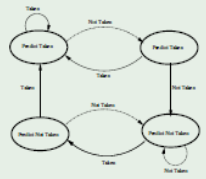
- The predictor must miss twice before the prediction changes.
- Conditional branching in regular recurring fashion goes undetected.

Two Level Adaptive Branch Prediction [12]



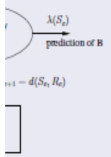


Abrupt incre
user due to



Dynamic 2-bit predictor State Machine

- The predictor must miss twice before the prediction changes.
- Conditional branching in regular recurring fashion goes undetected.



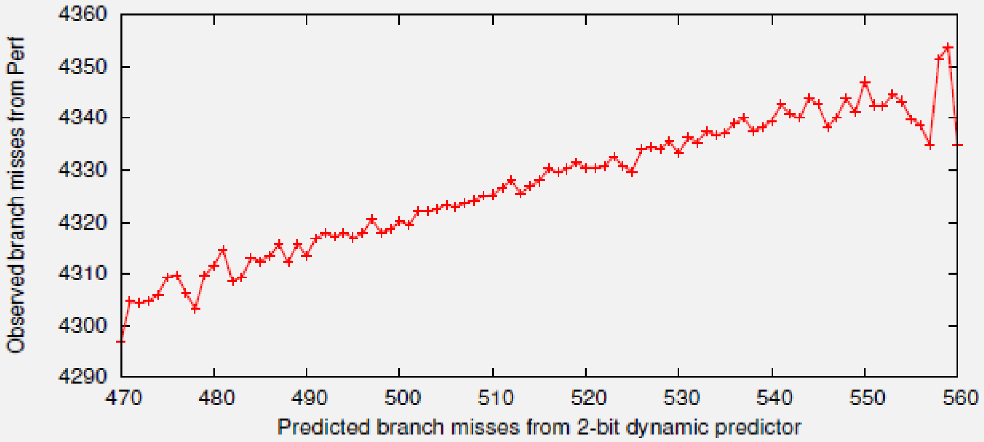
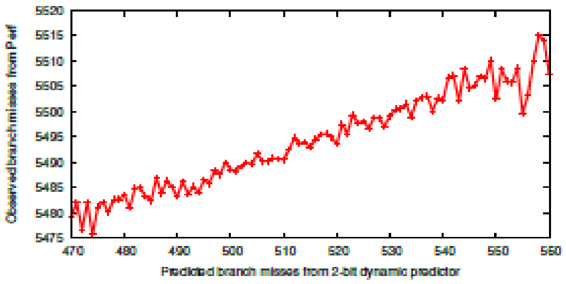
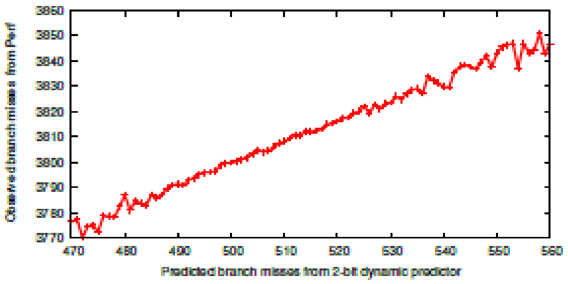


Figure: Variation of branch-misses from performance counters with increase in branch miss from 2-bit predictor algorithm

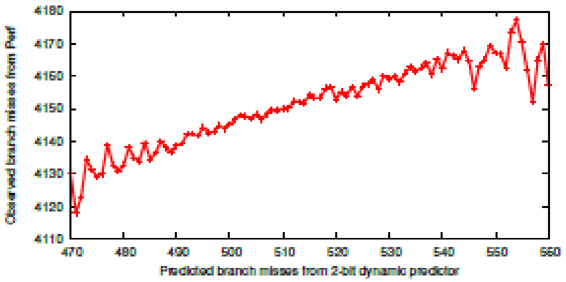
- Direct correlation observed for the branch misses from HPCs and from the simulated 2-bit dynamic predictor over a sample of exponent bitstream.
- This confirms assumption of 2-bit dynamic predictor being an approximation to the underlying system branch predictor.



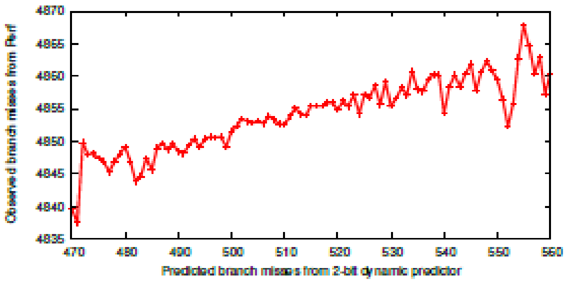
(a) Branch misses observed on Intel Core2 Duo-E7400



(b) Branch misses observed on Intel i3-M350



(c) Branch misses observed on Intel i5-3470



(d) Branch misses observed on AMD FX-8350

Variation of branch-misses from HPCs with increase in branch miss from 2-bit predictor algorithm on various platforms

Secret Dependent Branching

Let n-bit secret scalar in ECC be denoted as $(k_0, k_1, \cdots, k_i, \cdots, k_{n-1})$. Trace of taken or not-taken branches as conditioned on scalar bits and expressed as $(b_0, b_1, \cdots, b_{n-1})$.

- If a particular key bit k_j is 1 then the conditional addition statement in the double and add algorithm gets executed. Thus, the condition is checked first, and if the particular key bit is set then its immediate next statement ie, addition gets performed. Since this is a normal flow of execution the branch is considered as not-taken ie, $b_j = 0$ in this case.
- While when $k_j = 0$, the addition operation is skipped and the execution continues with the next squaring statement. Thus, in this case branch is taken ie, $b_j = 1$.

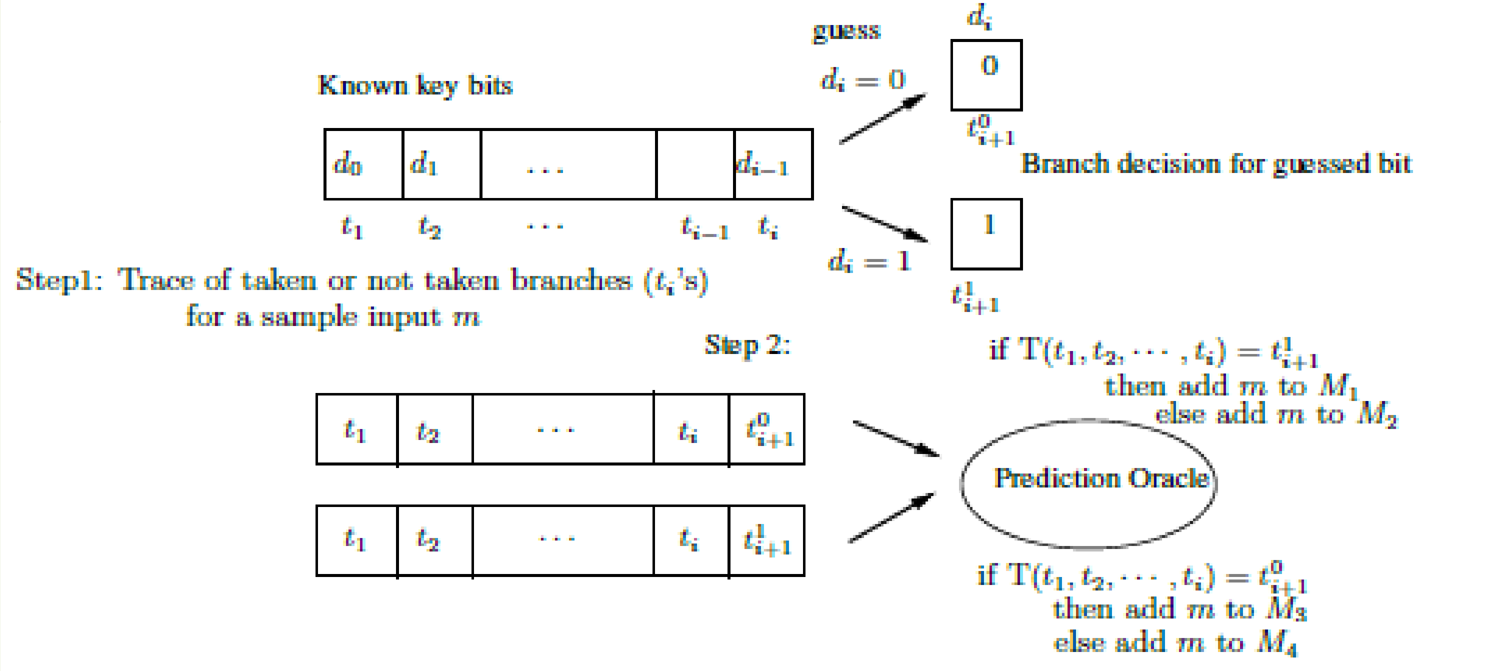
Effect of Compiler Optimization on branching

- We validate our understanding for conditional branching and observe the effect of optimization options in gcc:
  1. $.LC3 : .string\ hello$
  2. $.LC4 : .string\ hi$

without Optimization	O1	O2	O3

.L5:
movl −36(%rbp), %eax
cltq
movzbl −32(%rbp,%rax), %eax
cmpb $49, %al
jne .L3
movl $.LC3, %edi
call puts
jmp .L4
.L3:
movl $.LC4, %edi
call puts
	.L5:
	cmpb $49, (%rsp,%rbx)
	jne .L3
	movl $.LC3, %edi
	call puts
	jmp .L4
	.L3:
	movl $.LC4, %edi
	call puts
		.L3:
		movl $.LC4, %edi
		call puts
		.L5:
		----- 	
		jne .L3
		movl $.LC3, %edi
		---- 	
			.L3:
			movl $.LC4, %edi
			call puts
			.L5:
			--- 	
			jne .L3
			movl $.LC3, %edi
			call puts
			--- 	

Figure: Assembly generated using various optimization options in gcc



1. $M_1 = \{m | m$ does not cause a miss during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
2. $M_2 = \{m | m$ causes a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
3. $M_3 = \{m | m$ does not cause a miss during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$
4. $M_4 = \{m | m$ causes a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$

We ensure that there must be no common ciphertexts in sets (M_1, M_3) and (M_2, M_4) and the sets should be disjoint.

The probable next bit is decided by the following:

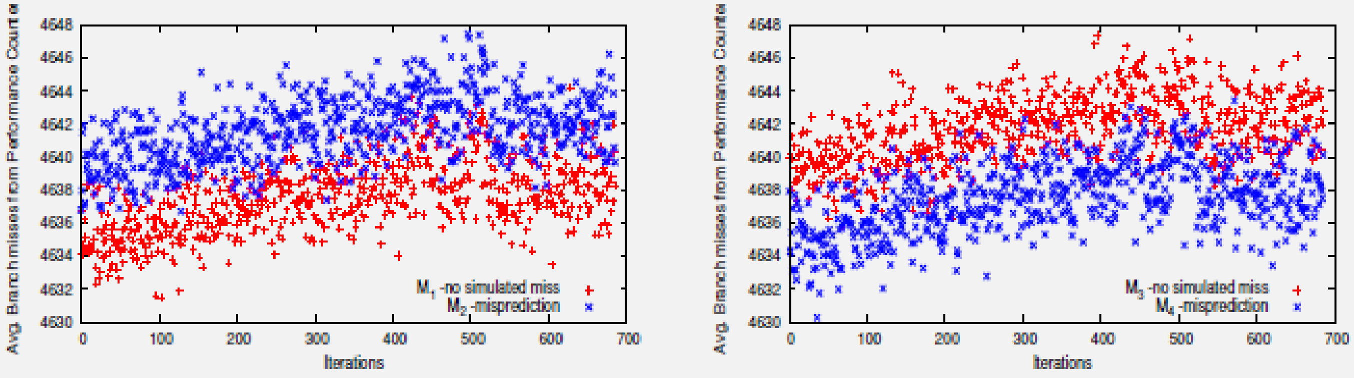
- If$(avg(\mathcal{M}_{M_2}) > avg(\mathcal{M}_{M_1}))$ and $(avg(\mathcal{M}_{M_4}) < avg(\mathcal{M}_{M_3}))$, then the next bit $(nb_i) = 1$
- Otherwise, if $(avg(\mathcal{M}_{M_4}) > avg(\mathcal{M}_{M_3}))$ and $(avg(\mathcal{M}_{M_2}) < avg(\mathcal{M}_{M_1}))$ then, next bit $(nb_i) = 0$

Algorithm 4: Adversary Attack Algorithm

```
Input: (d_0, d_1, ···, d_{i-1}), M
Output: Probable next bit nb_i
begin
    Offline Phase;
    for ∀m ∈ M do
        Generate taken/ not-taken trace for input m as t_{m,1}, t_{m,2}, ···, t_{m,i} ;
        Assume d_i = 0 and 1, generate t^0_{m,i+1}, t^1_{m,i+1} respectively;
        p_{m,i+1} = T(t_{m,1}, t_{m,2}, ···, t_{m,i}) ;
        if p_{m,i+1} = t^1_{m,i+1} then
            Add m to M_1 ;
        end
        else
            Add m to M_2 ;
        end
        if p_{m,i+1} = t^0_{m,i+1} then
            Add m to M_3 ;
        end
        else
            Add m to M_4 ;
        end
    end
    Remove Duplicate Ciphertexts in the sets M_1, M_3 and M_2, M_4;
    Online Phase;
    Observe distribution of branch misses from performance counters as 𝓜_{M_1}, 𝓜_{M_2}, 𝓜_{M_3}, 𝓜_{M_4} ;
    if (avg(𝓜_{M_2}) > avg(𝓜_{M_1})) and (avg(𝓜_{M_4}) < avg(𝓜_{M_3})) then
        nb_i = 1 ;
    end
    if (avg(𝓜_{M_4}) > avg(𝓜_{M_3})) and (avg(𝓜_{M_2}) < avg(𝓜_{M_1})) then
        nb_i = 0 ;
    end
    return nb_i ;
end
```

The probable next bit is decided by the following:

- If$(avg(\mathcal{M}_{M_2}) > avg(\mathcal{M}_{M_1}))$ and $(avg(\mathcal{M}_{M_4}) < avg(\mathcal{M}_{M_3}))$, then the next bit $(nb_i) = 1$
- Otherwise, if $(avg(\mathcal{M}_{M_4}) > avg(\mathcal{M}_{M_3}))$ and $(avg(\mathcal{M}_{M_2}) < avg(\mathcal{M}_{M_1}))$ then, next bit $(nb_i) = 0$



(a) **Correct Assumption** $d_i = 1$ (b) **Incorrect Assumption** $d_i = 0$

Figure: Branch misses from HPCs on square and multiply correctly identifies secret bit $d_i = 1$, ciphertext set partitioned by simulated misses of two-level adaptive predictor

Al

Input

Outp

begi

Observe distribution of branch misses from performance counters as $\mathcal{M}_{M_1}, \mathcal{M}_{M_2}, \mathcal{M}_{M_3}, \mathcal{M}_{M_4}$;

if $(avg(\mathcal{M}_{M_2}) > avg(\mathcal{M}_{M_1}))$ *and* $(avg(\mathcal{M}_{M_4}) < avg(\mathcal{M}_{M_3}))$ **then**

  $nb_i = 1$;

end

if $(avg(\mathcal{M}_{M_4}) > avg(\mathcal{M}_{M_3}))$ *and* $(avg(\mathcal{M}_{M_2}) < avg(\mathcal{M}_{M_1}))$ **then**

  $nb_i = 0$;

end

return nb_i ;

end

Observation 2

Reverse Engineering of Branch Prediction

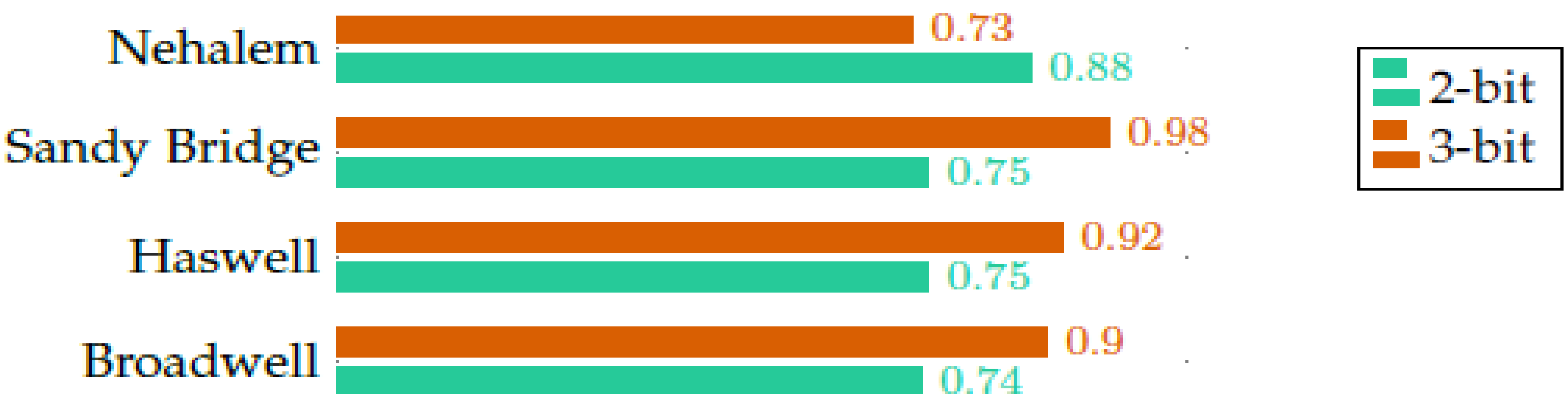
1. We perform a reverse engineering of the branch predictor hardware and found that the behavior has a significantly high correlation to the deterministic 3-bit predictor characteristics.

Branch prediction hardware design is proprietary of the processor manufacturer.

- The perf class is instantiated with particular hardware event.
- We incorporate start and stop calls before and after the target conditional if-else structure.
- This returns event counts at regular interval and measurements are synchronous to the execution of the conditional block.

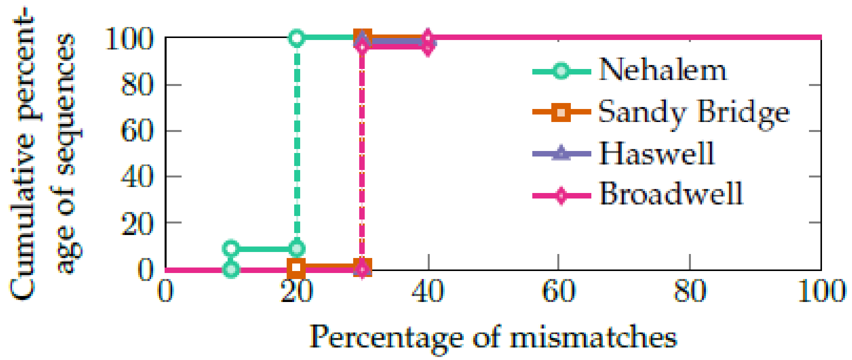
```
static long
  perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                  int cpu, int group_fd, unsigned long flags)
  {
     int ret;

     ret = syscall(__NR_perf_event_open, hw_event, pid, cpu,
                   group_fd, flags);
     return ret;
  }
   void start()
  {
    int rc = ioctl(fd_, PERF_EVENT_IOC_RESET, 0);
    assert(rc == 0);
    rc = ioctl(fd_, PERF_EVENT_IOC_ENABLE, 0);
    assert(rc == 0);
  }
  size_t stop()
  {
    int rc = ioctl(fd_, PERF_EVENT_IOC_DISABLE, 0);
    assert(rc == 0);
    size_t count;
    int got = read(fd_, &count, sizeof(count));
    assert(got == sizeof(count));
    return count;
  }
```

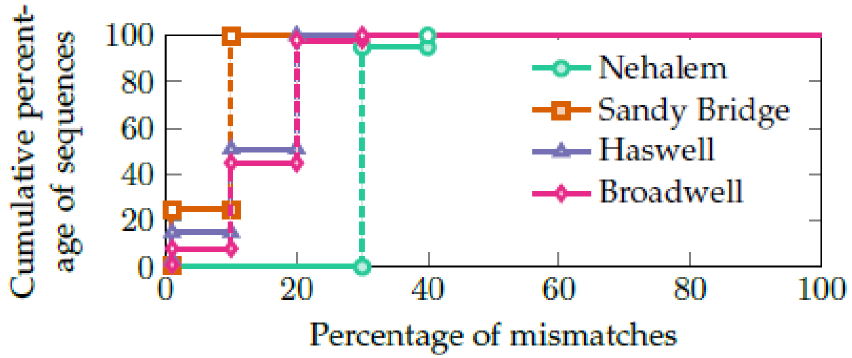


: Model accuracy on average for the 2-bit and 3-bit sat-
ig counter state machines, for four micro-architectures.

[Predictor model: 2-bit saturating counter state machine]



[Predictor model: 3-bit saturating counter state machine]



Deduce & Remove Attack on Blinded Scalar Multiplication with Asynchronous perf ioctl Calls

Overview

- HPCs are potential side channel source for implementations using conditional branching where the hardware is typically shared between multiple users.
- However, existing research considers blinding techniques, like scalar blinding, scalar splitting as a mechanism of thwarting such attacks.
- We reverse engineer the undisclosed model of Intel's Broadwell and Sandybridge branch predictor and further utilize the unexplored perf ioctl calls in sampling mode to granularly monitor the branch prediction events asynchronously when a victim cipher is executing.

Objective

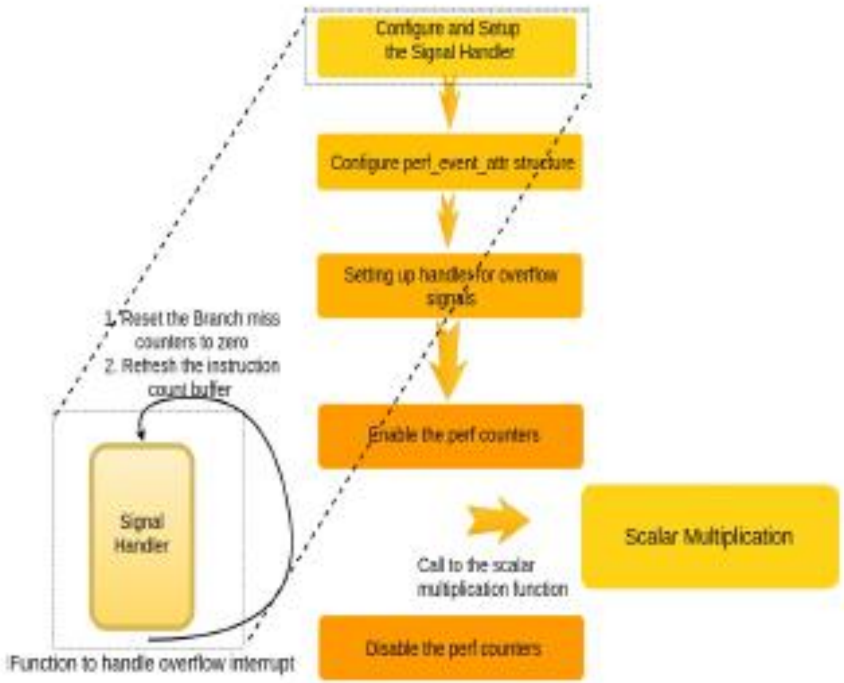
- We target the harder problem of attacking the DPA secure implementations such as scalar splitting and scalar blinding using the perf ioctl system calls.
- The samples obtained are inherently noisy because of its asynchronous nature.
- Traces obtained lack proper synchronization and measurements at regular time-step.
- The target algorithm being randomized in nature adds to the difficulty of attacking with such coarse measurements.

Principle

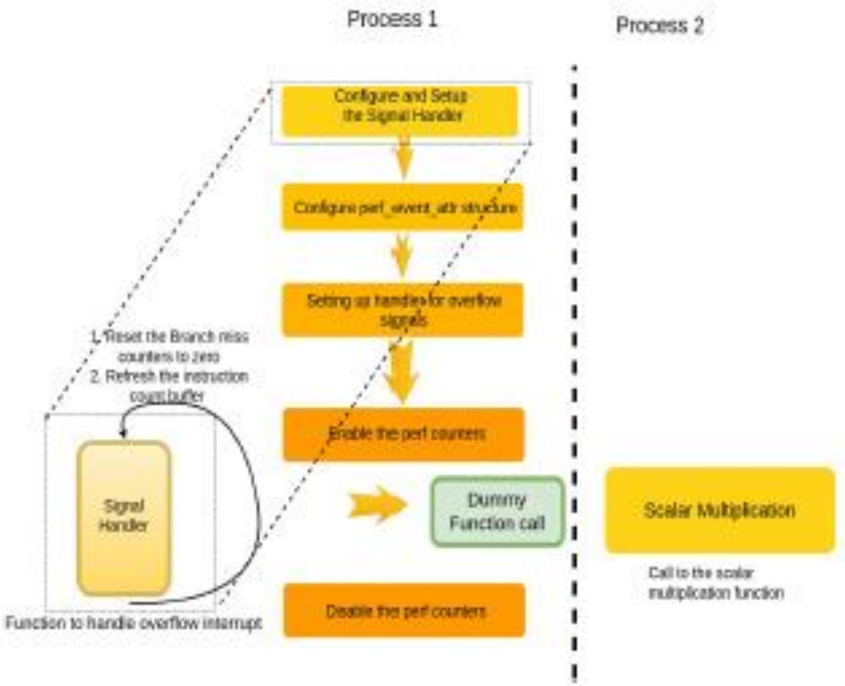
Thus we follow by a principle of,

- *Acquire*: obtain branch misprediction traces over the scalar multiplication.
- *Deduce*: every randomized trace should reveal partial key bits.
- *Remove*: if a randomized trace does not leak any information regarding the trace, then the attacker should be able to isolate and remove the trace.

Scenarios



(a) Scenario using Perf sampler in asynchronous sampling mode



(b) Using Perf sampler in asynchronous sampling mode from two different scripts

Understanding Branch Mispredictions Existing DPA countermeasures on ECC

Scalar Randomization[1]

If K is the secret scalar and $P \in E$ the base point, instead of computing K times P, randomize the scalar K as $K' = K + r * \#E$ where r is a random integer and $\#E$ is the order of the curve.

Scalar Splitting

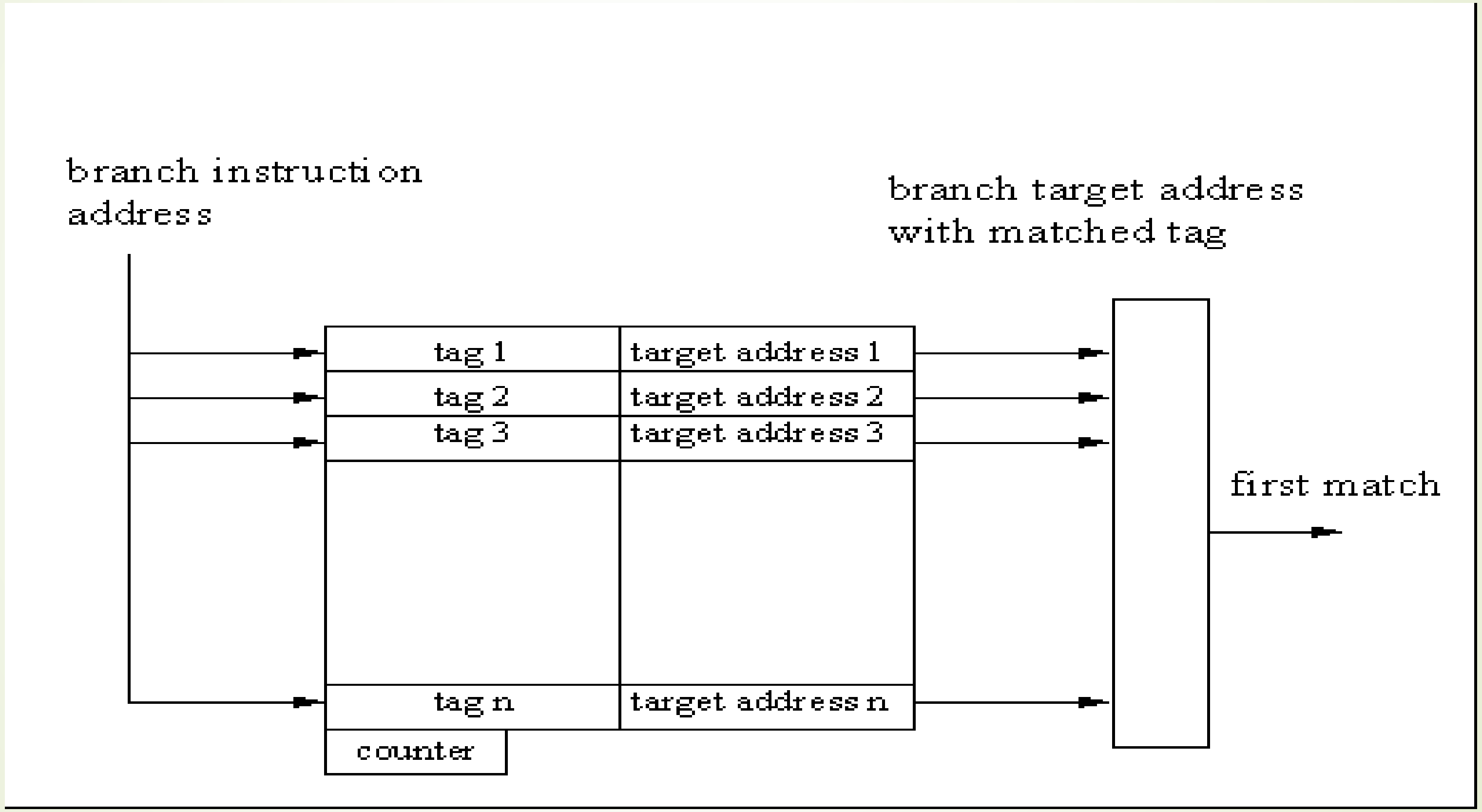
In [2], to randomize the scalar such that instead of computing KP, the scalar is split in two parts $K = (K - r) + r$ with a random r, and multiplication is computed separately, $KP = (K - r)P + rP$.

Point Blinding

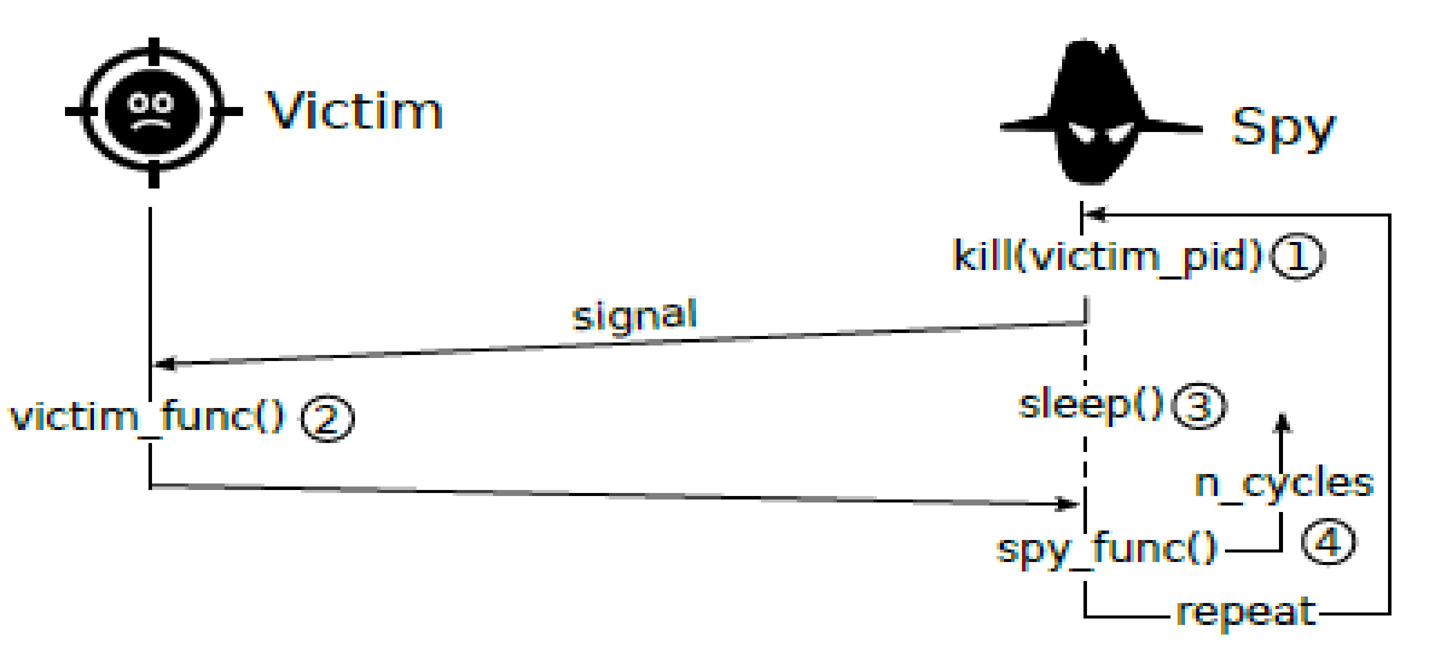
This computes $K(P + R)$ instead of KP, where KR can be stored in the system beforehand, which when subtracted $K(P + R) - KR$ gives back KP.

Observation 3

BTB structure and Collision



Threat Model



Spy

Always executes taken branch which jumps to target address TA_spy

Could execute a taken or not-taken branch

Victim

Measure

Measures the access time of the target address TA_spy

Secure Predictors

Fortifying Branch Predictors to thwart Micro-architectural Attacks

Contributions

1. The primary contribution of this work is a secure design of branch predictor: λ-confidence predictor which invalidates the direct proportionality of branch mispredictions from known predictor structures.

2. A hashed indexing scheme which is essential to prevent branch collision based attacks on the shared table structures such as BTB and PHT.

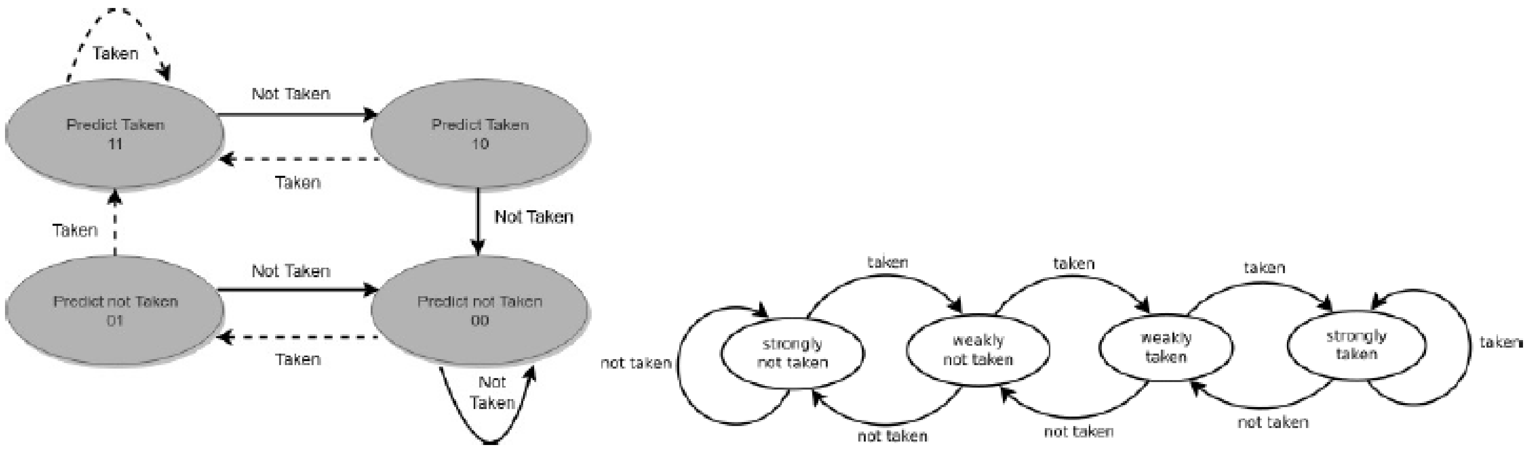
3. Performance comparison of the new predictor to state-of-art predictors like Gshare and more recent TAGE-based predictors using traces from SPEC-2006, server and multimedia benchmarks in terms of MisPredictions per Kilo Instructions (MPKI) and misprediction penalty, to demonstrate that the design do not compromise on performance.

4. Lastly, test for security on cryptographic implementations and the design has lesser information leakage than predictors in literature.

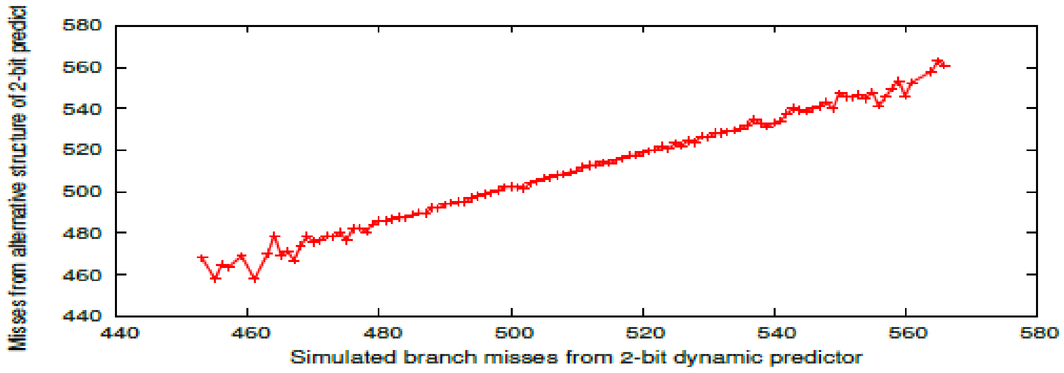
Why is it important to design secure branch predictors ?

“Does making cryptographic implementations free from conditional branching totally do away with the threat of micro-architectural attacks caused due to the branch predictors?”

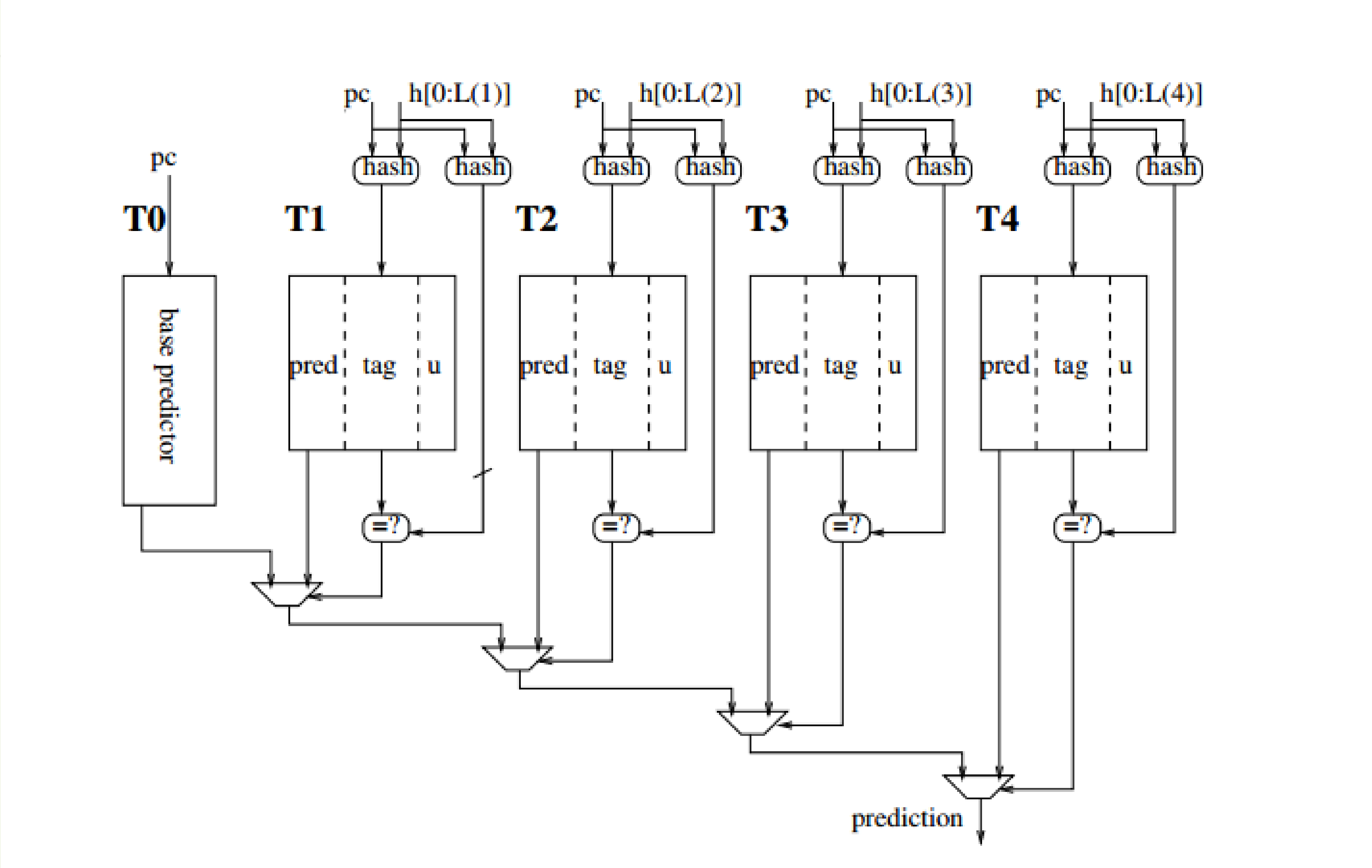
Insecurity of Commercial Intel Systems



(a) 2-bit dynamic predictor (HP06). (b) Alternative structure of 2-bit dynamic predictor (HP06).



InSecurity for TAGE based predictor structures



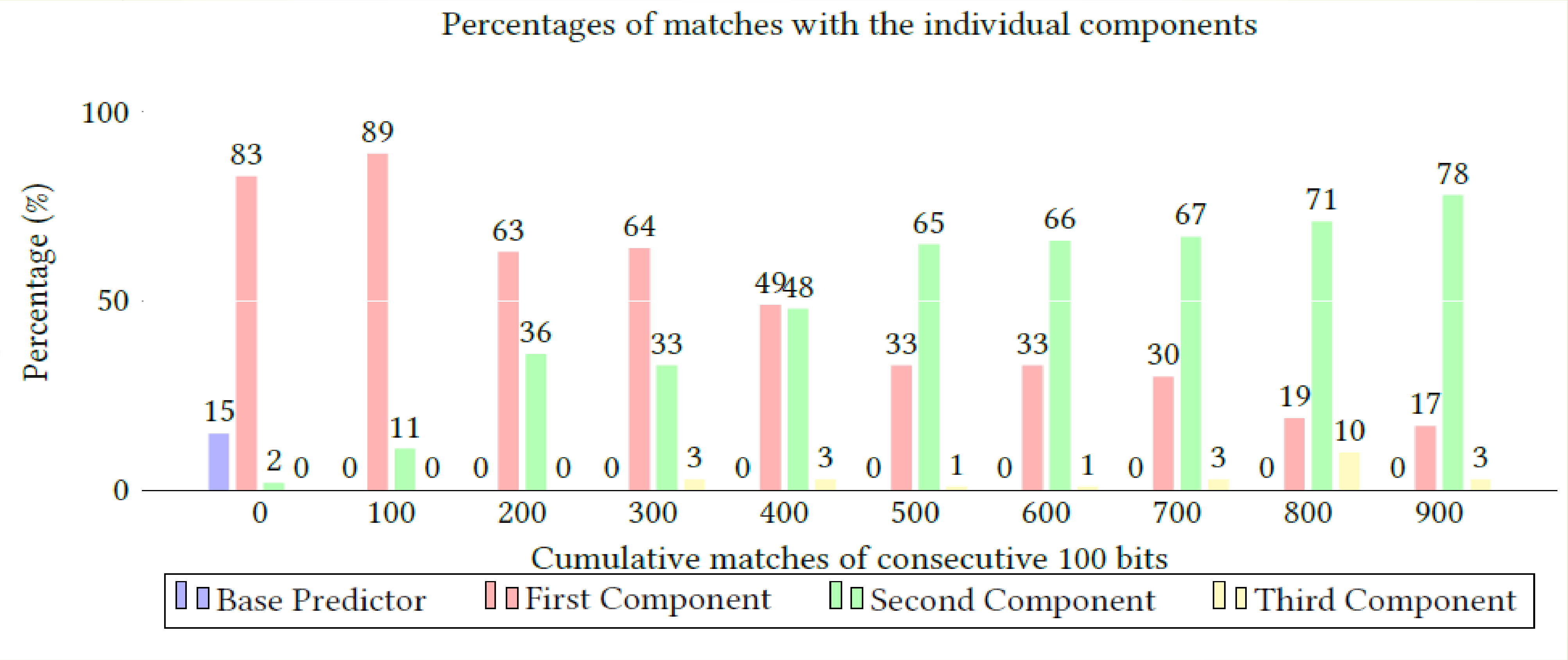
How are TAGE predictors vulnerable

- Initial branches, where there are no matched tags in any of the component structures: This is the phase when the execution just starts and the index decided by the program counter xored with none of the previous history, do not match with any stored tags in the computed indices. In this part of the execution, the predictions are made using the base predictor.

- When there are some tags which match the existing history based tags and some does not. In this case, the 3-bit predictor of the first component table and the base predictor are most likely to provide the prediction.

- Third case arises, for the final bits of execution which shows tag, index match in multiple component tables of TAGE. But the final prediction is such that in each of these component tables the 3-bit predictors provide the final prediction.

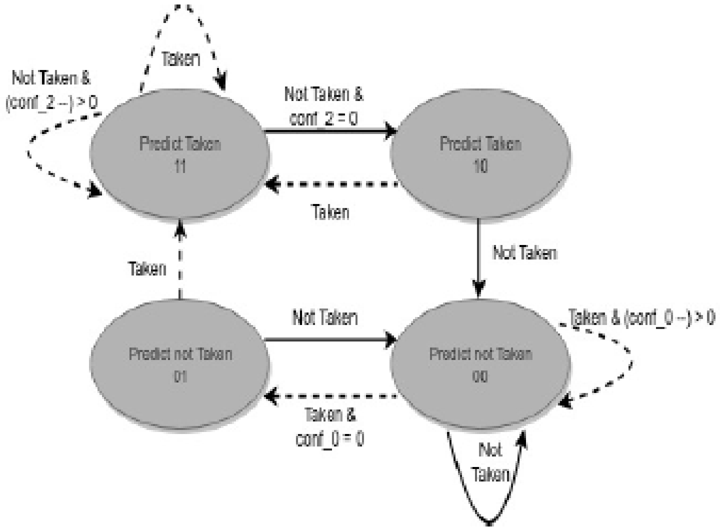
How are TAGE predictors vulnerable



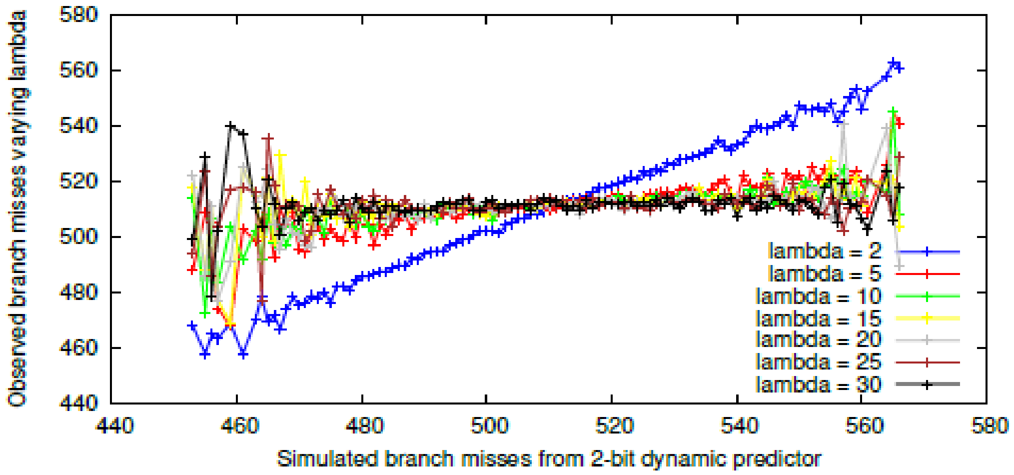
Aim of λ Predictor: Performance + Security

- Performance for Benchmark Programs
- Security for Sensitive Applications

Adding Lambda confidence to generic predictor model

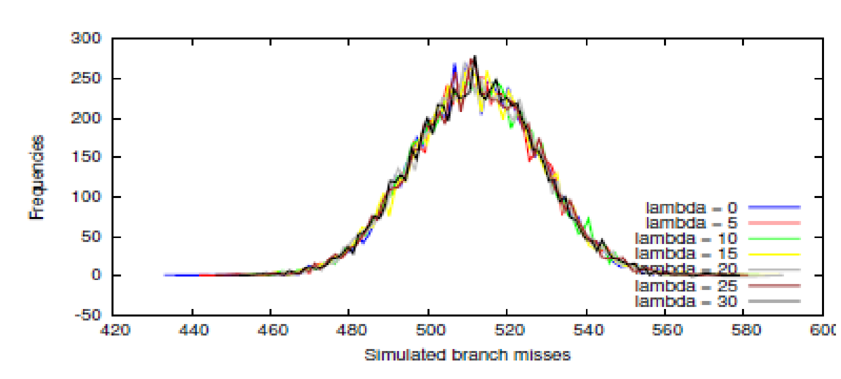


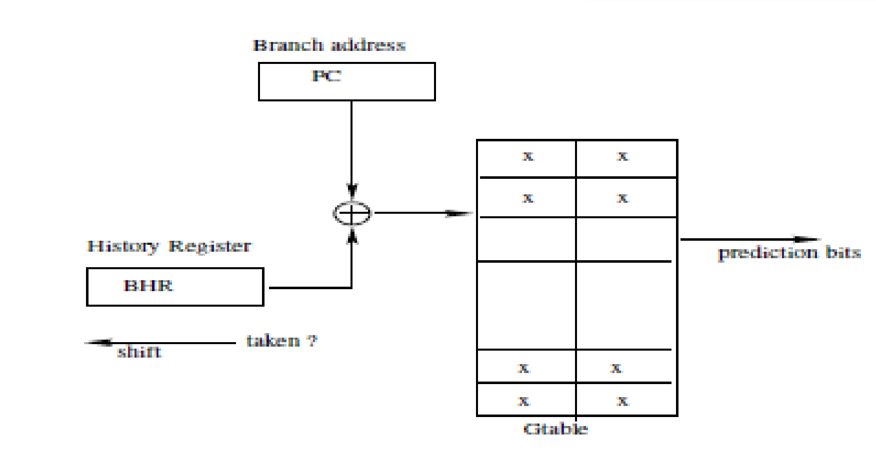
(a) λ-confidence predictor.

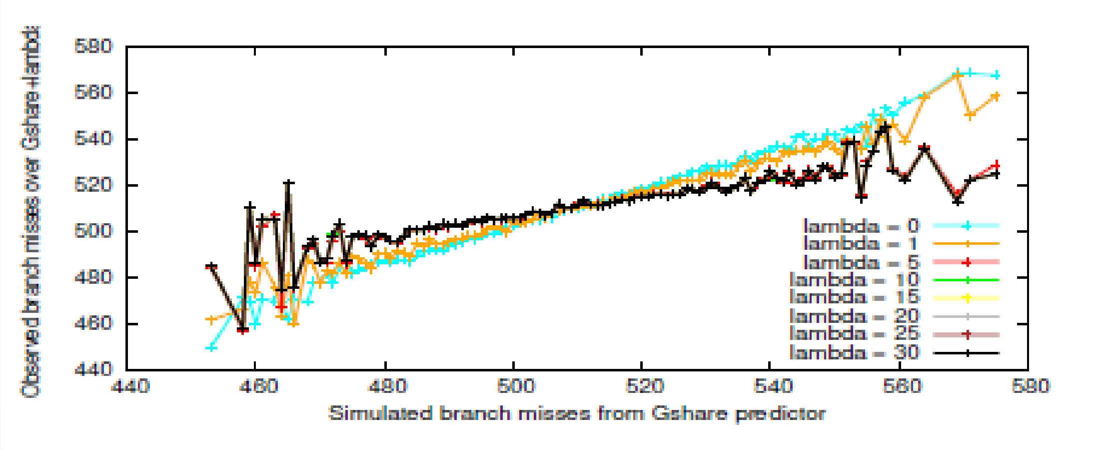


(b) Branch misses plotted for various values of λ across misses from 2-bit branch predictor.

Effect of Lambda on Gshare predictors

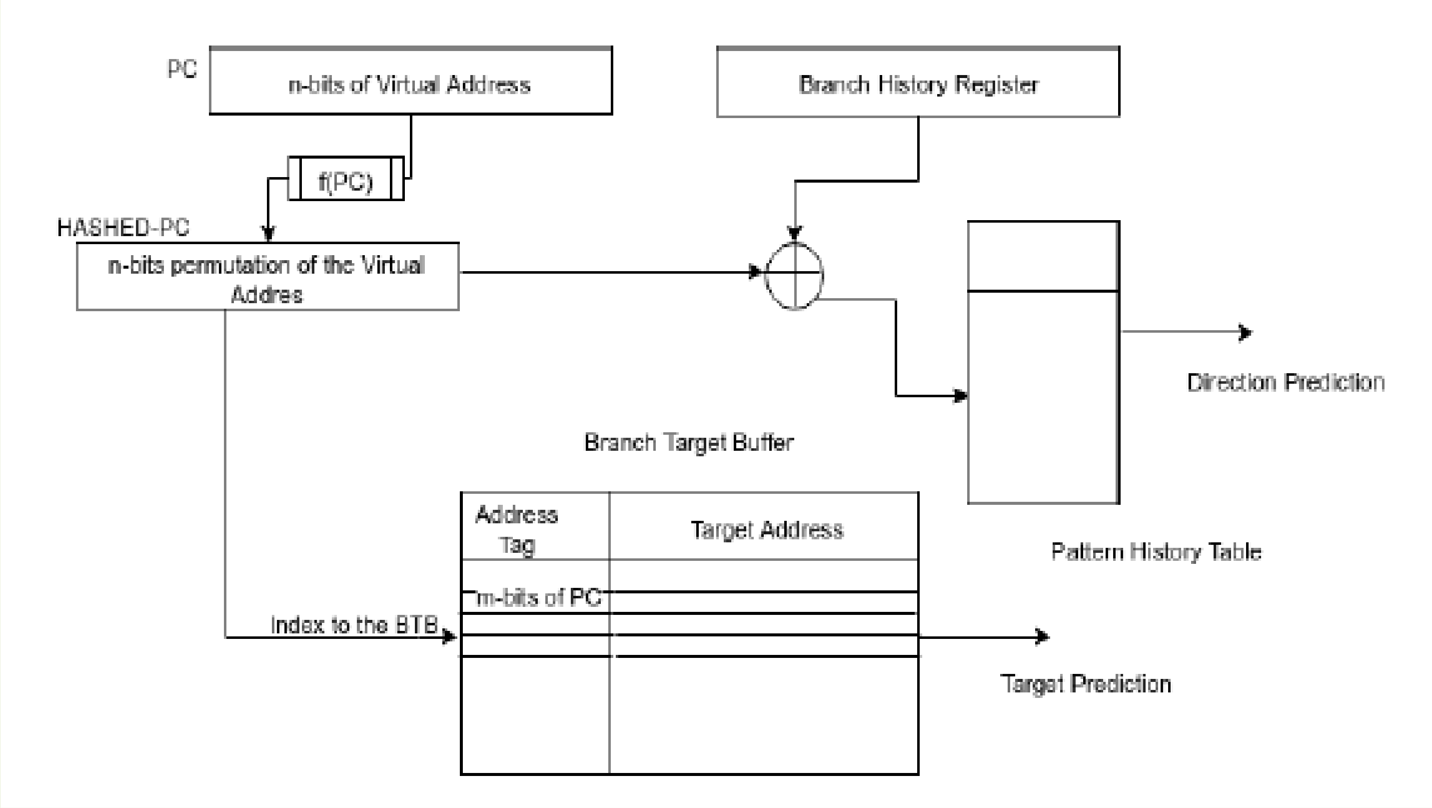




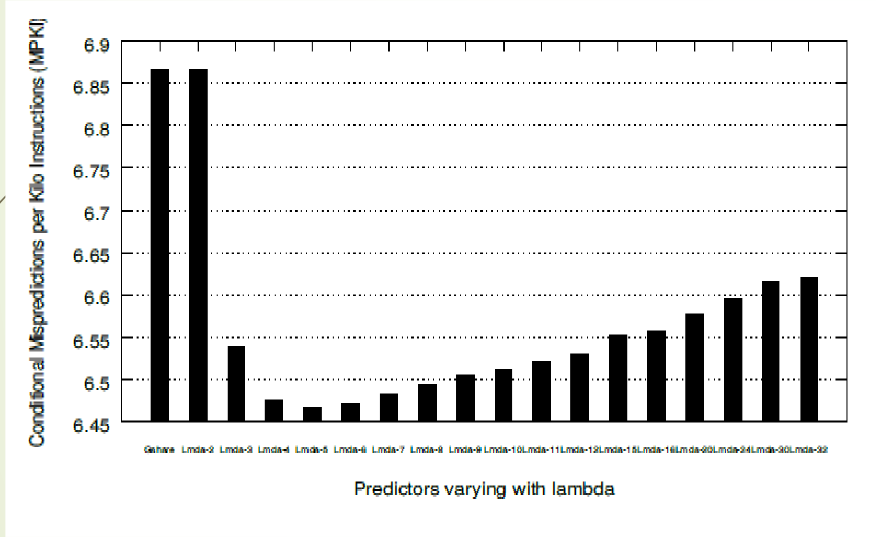


Branch misses plotted for various values of λ across misses from Gshare structure.

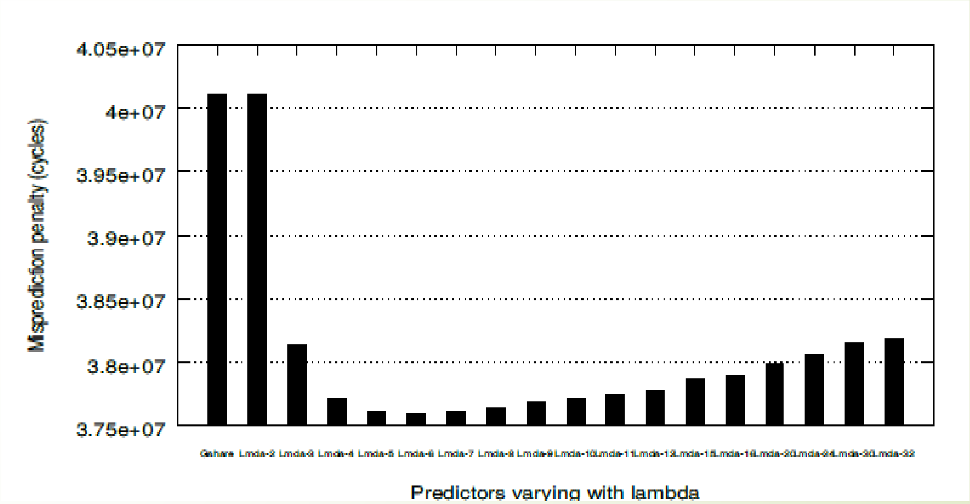
Securing BTB from collision based attacks



Results on values of Lambda

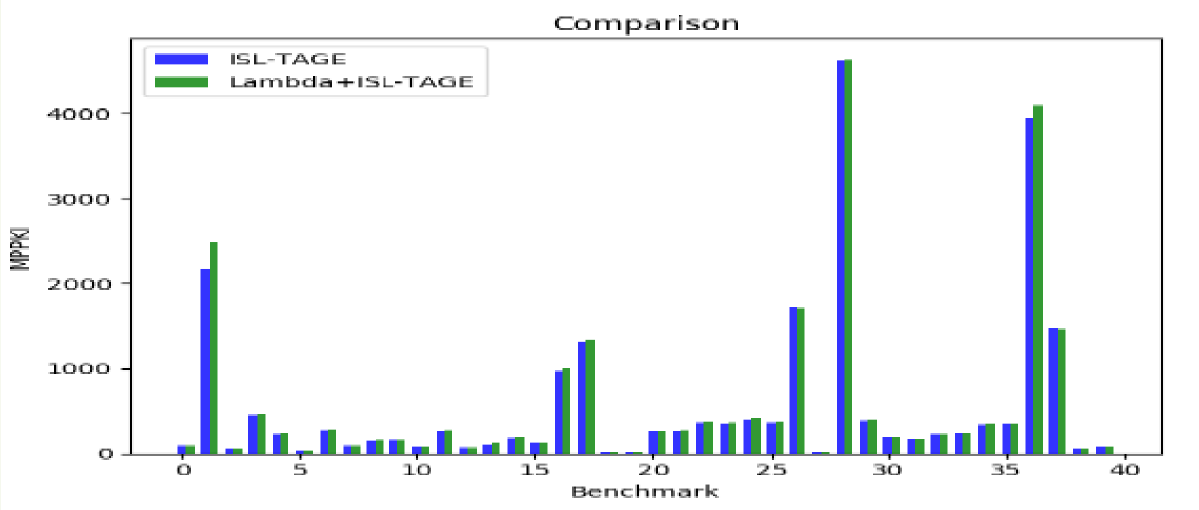


MPKI



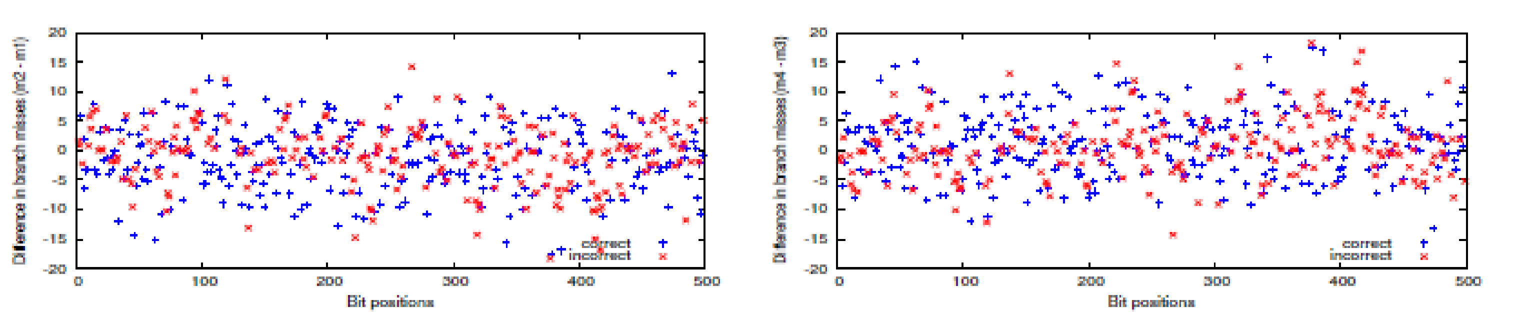
Misprediction Penalty

Performance of λ + ISL-TAGE



Performance of λ+ISL-TAGE predictor on CBP3 Benchmark

Inconclusive DOM results after introducing λ



(a) Branch misses for guess 0. (b) Branch misses for guess 1.

Branch Prediction Attack on RSA-OAEP Randomized Padding Scheme

THANK YOU FOR YOUR ATTENTION!