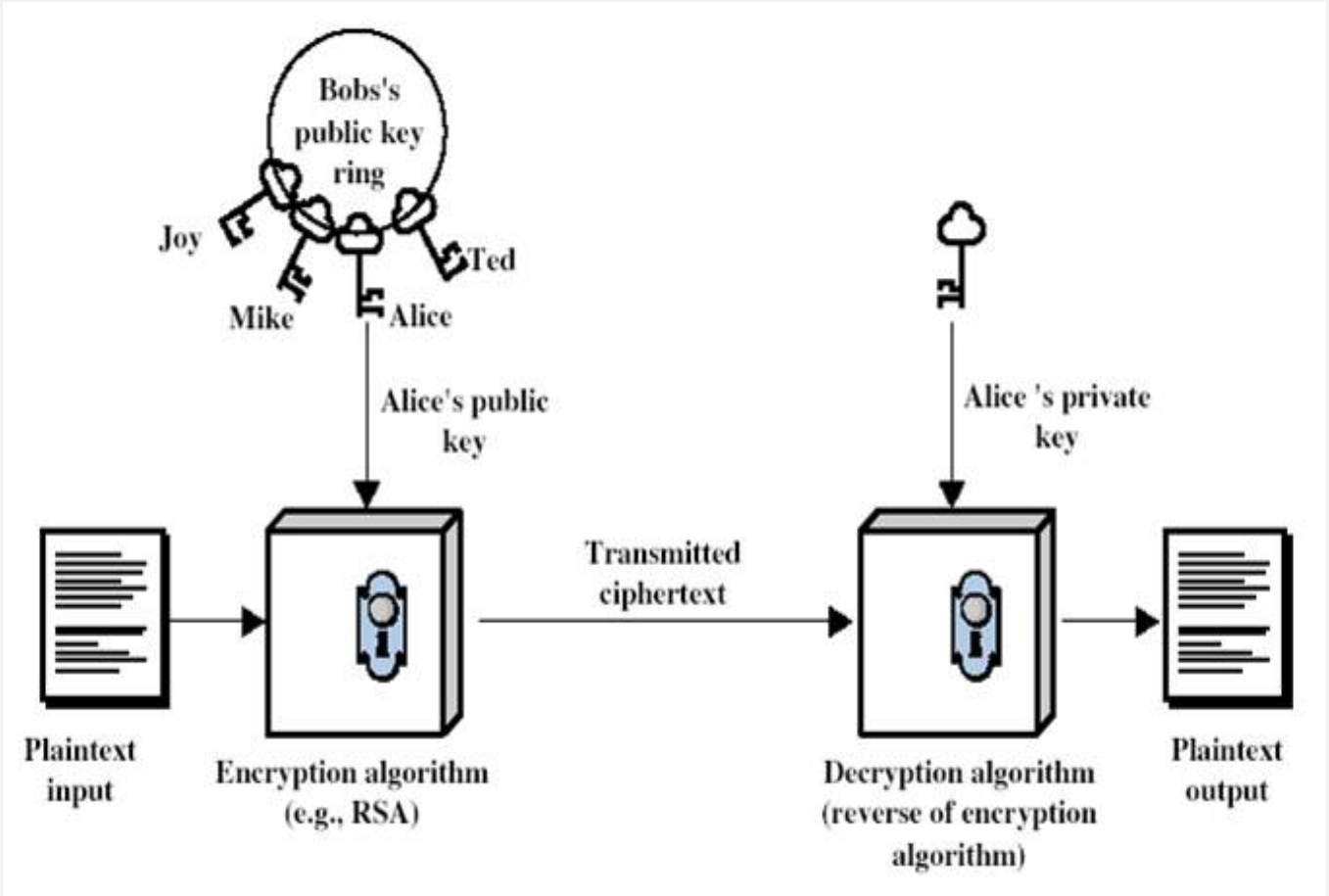
Branch Prediction based attacks using Hardware performance Counters

IIT Kharagpur

March 19, 2018

Public key Cryptography



Exponentiation and Underlying Multiplication Primitive

- Inputs(M) are encrypted and decrypted by performing modular exponentiation with modulus N on public or private keys represented as n bit binary string.

Algorithm 1: Binary version of Square and Multiply Exponentiation Algorithm

$S \leftarrow M$;
for i **from** 1 **to** $n-1$ **do**
  $S \leftarrow S * S \ mod \ N$;
  if $d_i = 1$ **then**
    $S \leftarrow S * M \ mod \ N$;
  end
end
return S ;

- Conditional execution of instruction and their dependence on secret exponent is exploited by the simple power and timing side-channels.

Montgomery Ladder Exponentiation Algorithm

- A naïve modification is to have a balanced ladder structure having equal number of squarings and multiplications.
- Most popular exponentiation primitive for Asymmetric-key cryptographic implementations.

Algorithm 2: Montgomery Ladder Algorithm

$R_0 \leftarrow 1$;
$R_1 \leftarrow M$;
for i **from** 0 **to** $n-1$ **do**
    if $d_i = 0$ **then**
        $R_1 \leftarrow (R_0 * R_1) \mod N$;
        $R_0 \leftarrow (R_0 * R_0) \mod N$;
    end
    else
        $R_0 \leftarrow (R_0 * R_1) \mod N$;
        $R_1 \leftarrow (R_1 * R_1) \mod N$;
    end
end
return R_0 ;

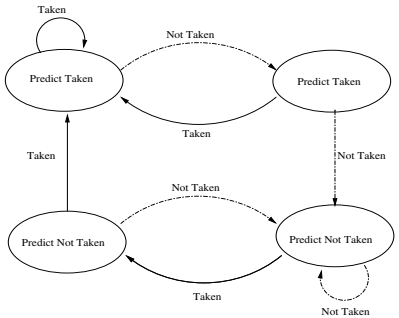
Montgomery Multiplication Algorithm

- Highly efficient algorithm for performing modular squaring and modular multiplication operation.
- Avoids time consuming integer division operation.
- R is assumed to be 2^k, when N is k-bit number.
- Calculates $Z = A * B * R^{-1}(\text{mod}N)$, $A = a * R(\text{mod}N)$, $B = b * R(\text{mod}N)$ and $R^{-1} * R = 1(\text{mod}N)$.

Algorithm 3: Montgomery Multiplication Algorithm

$S \leftarrow A * B$;
$S \leftarrow (S + (S * N^{-1} \ mod \ R) * N)/R$;
if $S > N$ **then**
    $S \leftarrow S - N$;
end
return S ;

Branch Predictor State Machines



Dynamic 2-bit predictor State Machine

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

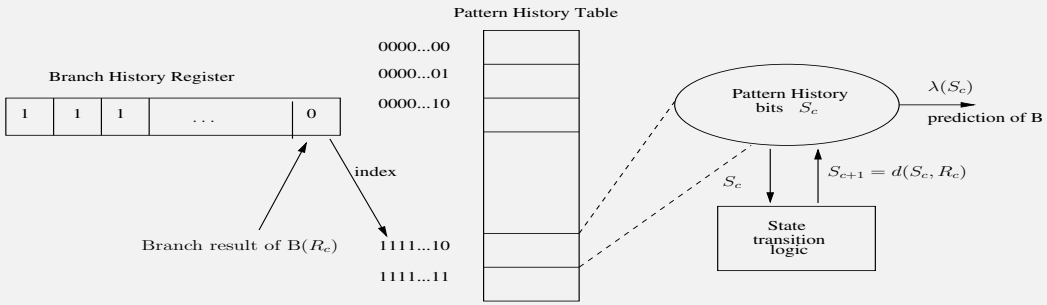


Figure: Two Level Adaptive Branch Prediction

Modelling Branch Miss as Side-Channel from HPC

- We monitor the branch misses on the square and multiply and Montgomery Ladder algorithm using Montgomery multiplication as subroutine for operations like squaring and multiplication.
- Branch miss rely on the ability of branch predictor to correctly predict future branches to be taken.
- Profiling of HPCs using performance monitoring tools provides simple user interface to different hardware event counts and are considered as side-channel.

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 Optimizations 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

Approximating the System predictor with 2-bit branch predictor

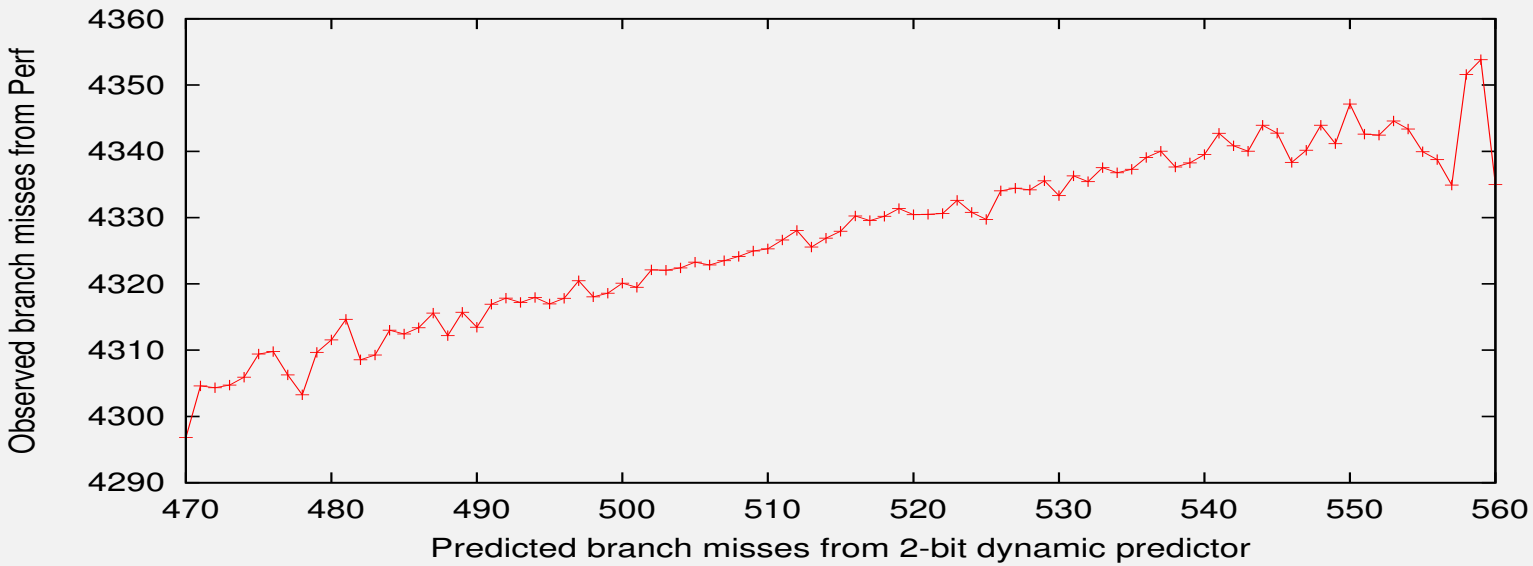


Figure: Variation of branch-misses from performance counters with increase in branch miss from 2-bit predictor algorithm

- 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.

Idea of the Attack

- Timing attack exploiting branch mispredictions are demonstrated which requires the knowledge of actual structure of branch prediction hardware of the target system.
- Advantage of this attack lies in the fact that adversary, inspite of having no knowledge of the underlying architecture, can actually target real systems and reveal secret exponent bits, exploiting the branch miss as side-channel from HPCs.
- This is an iterative attack, targeting i^{th} bit assuming previous bits to be known.
- The attack separates a sample input set based on mispredictions for conditional reduction of Montgomery multiplication at the $(i+1)^{th}$ squaring step of exponentiation assuming secret i^{th} bit.

Threat Model for the attack

- The attacker knows first i bits of the private key and wants to determine next unknown bit d_i of the key $(d_0, d_1, \cdots, d_i, \cdots, d_{n-1})$.
- Generate a trace of branches as $(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$ for conditional reduction of Montgomery multiplication at every squaring step.
- Under the assumption of d_i having value j, where $j \in \{0, 1\}$, appropriate value of $t^j_{m,i+1}$ is simulated.

Offline Phase of the Attack

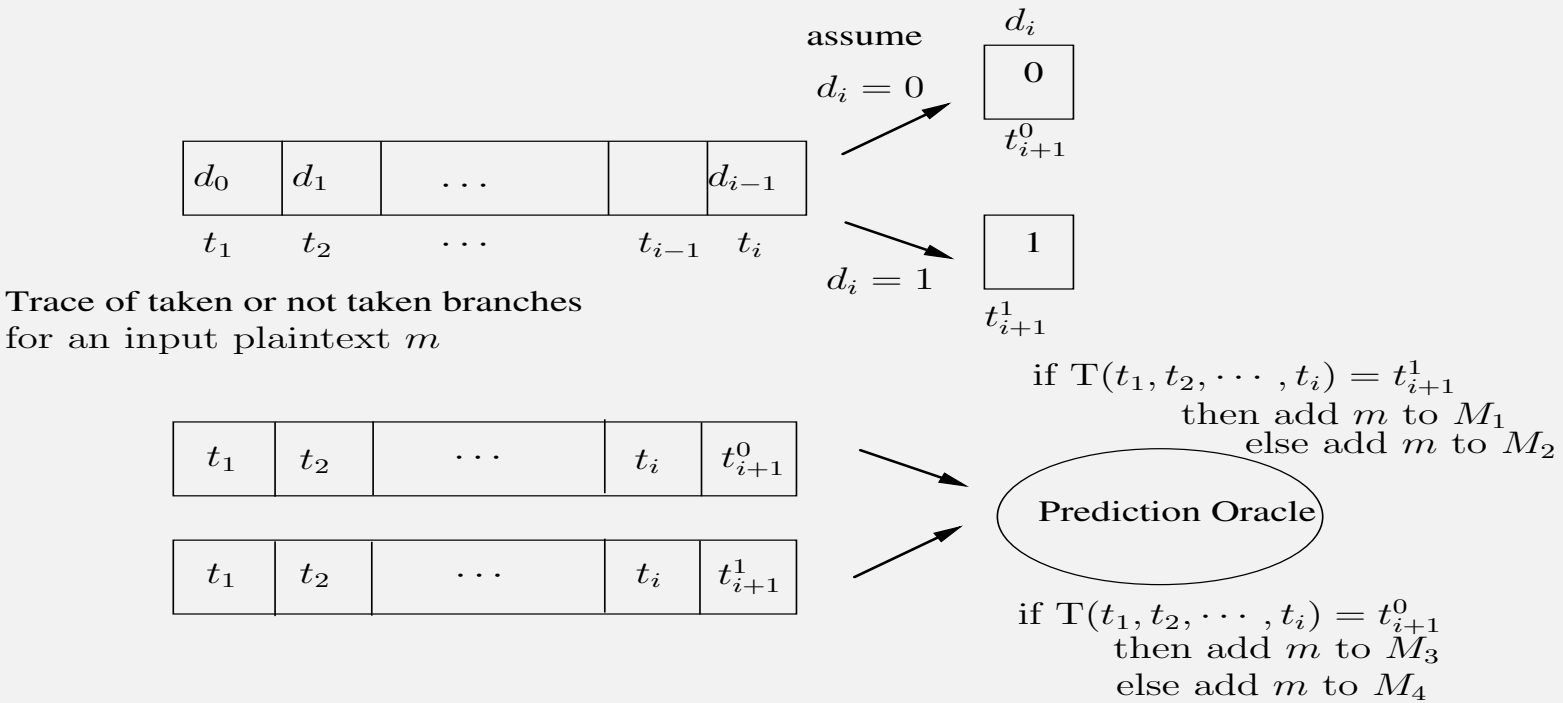


Figure: Partitioning randomly generated Ciphertexts set based on simulated Branch miss Modelling

Separation of Random Inputs

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.

Online Phase

The probable next bit is decided following the Algorithm 4.

- 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: $(d_0, d_1, \cdots, d_{i-1})$,$M$

Output: Probable next bit nb_i

begin

Offline Phase;

for $\forall m \in M$ **do**

Generate taken/ not-taken trace for input m as $t_{m,1}, t_{m,2}, \cdots, t_{m,i}$;

Assume $d_i = 0$ and 1, generate $t^0_{m,i+1}$, $t^1_{m,i+1}$ respectively;

$p_{m,i+1}$ = $T(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$;

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 $\mathcal{M}_{M_1}, \mathcal{M}_{M_2}, \mathcal{M}_{M_3}, \mathcal{M}_{M_4}$;

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

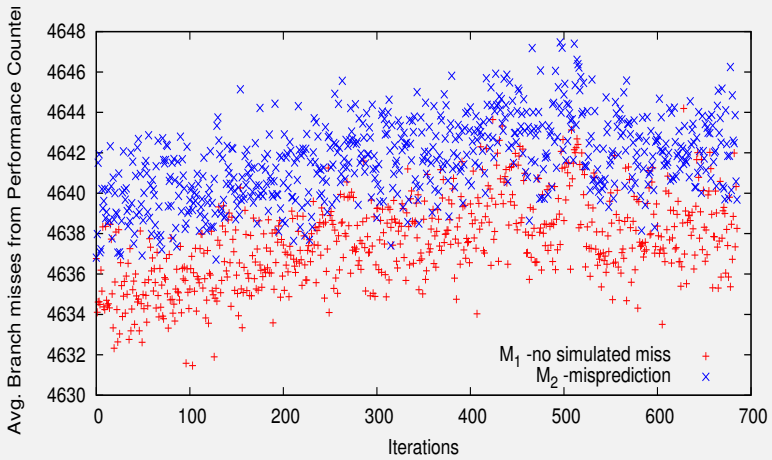
$nb_i = 1$;

end

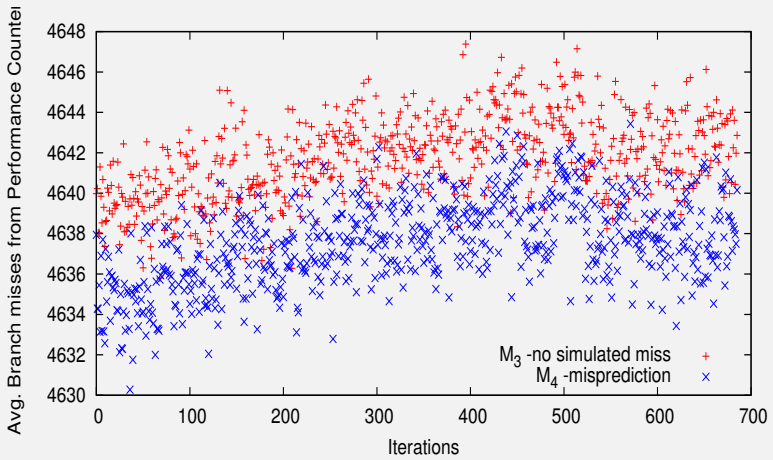
Experimental Validation for the Online Phase of the Attack

- A large input set is separated by simulations over bimodal and two-level adaptive predictor.
- Average branch misses are observed from HPCs for each elements in set M_1 , M_2 , M_3 and M_4.
- Each set has $L = 1000$ elements.
- Experiment is repeated over $I = 1000$ iterations.
- Experiments are performed on various platforms as Core-2 Duo E7400, Intel Core i3 M350 and Intel Core i5-3470.

Experiments on Square and Multiply Algorithm



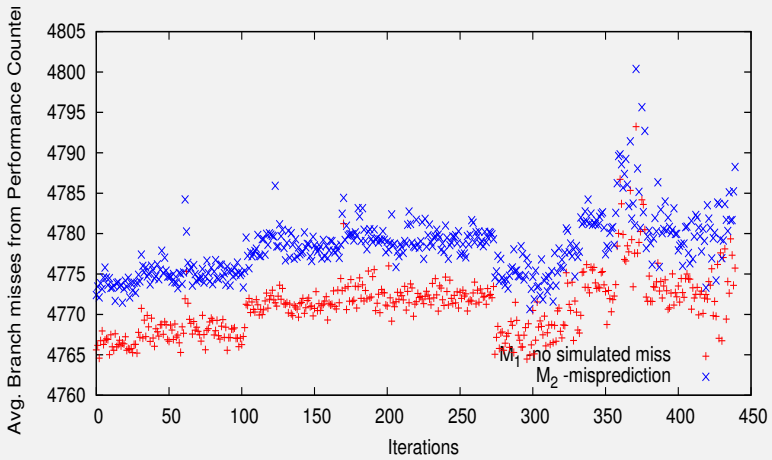
(a) **Correct Assumption** $d_i = 1$



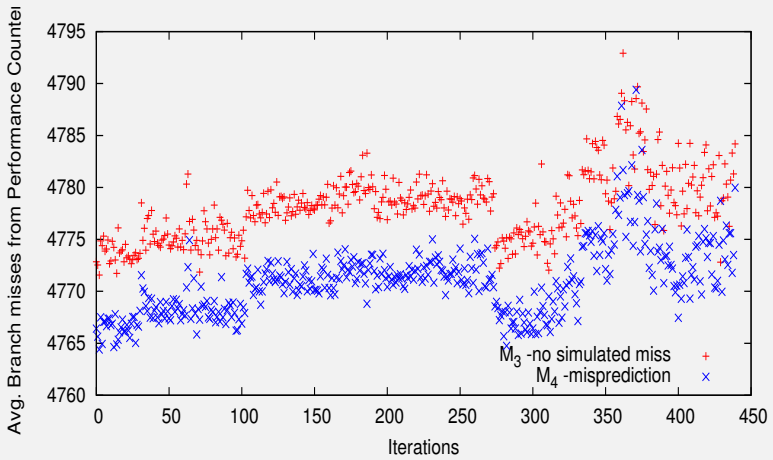
(b) **Incorrect Assumption** $d_i = 0$

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

Experiments on Montgomery Ladder



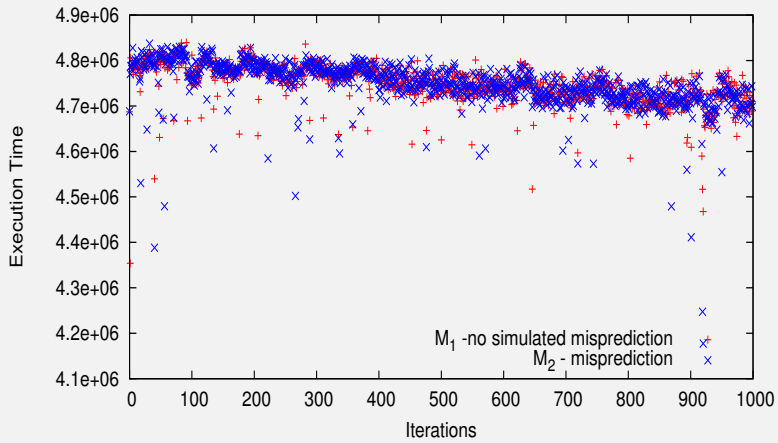
(a) **Correct Assumption** $d_i = 1$



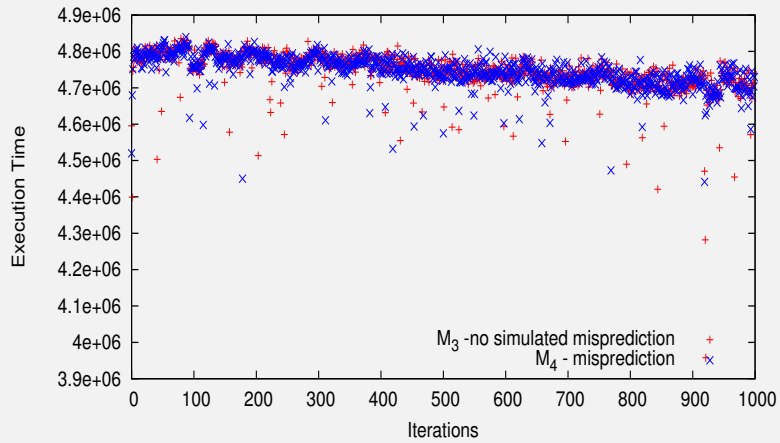
(b) **Incorrect Assumption** $d_i = 0$

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

Comparison with timing as side-channel



(a) **Correct Assumption** $d_i = 1$



(b) **Incorrect Assumption** $d_i = 0$

Figure: No identification of secret bit is possible using timing as side-channel with $L = 1000$ and $I = 1000$

Existing DPA countermeasures on ECC

Scalar Randomization

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.

Code Snippet for granular observations

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;
  }
```

Sampling granularly using Ioctl system call

- Previous measurements are not practical for an attacker, as the attacker cannot modify the executable run by the victim.
- Instead we use perf ioctl in sampling mode.
- We define a function as `signal handler` which execute on an interrupt raised by the interrupt handler.
- `perf` object is instantiated with an event that is used as sampler object with a predefined `sample_period`.
- The interrupt handler is called at regular intervals of the `sample_period`.
- Measurements are observed to be more noisy if `sample_period` is reduced beyond a threshold, due to the overhead of the interrupt getting generated very frequently.

Reverse Engineering of Branch Predictors in Modern Intel Processors

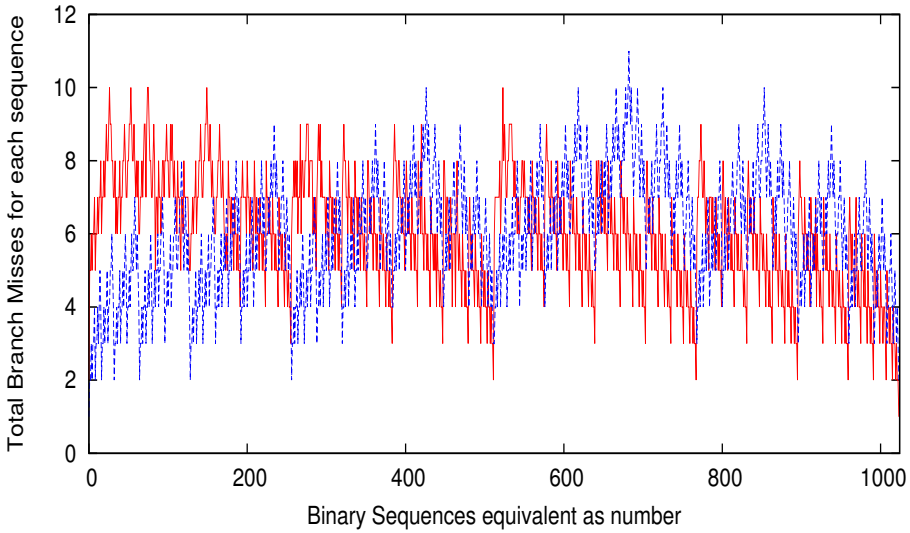


Figure: Branch misses from 2-bit predictor and actual hardware predictor

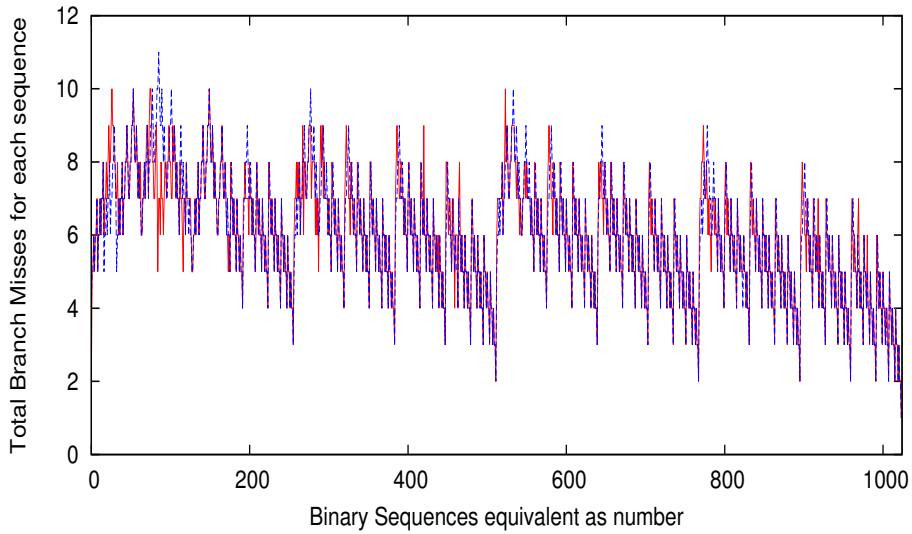


Figure: Branch misses from 3-bit predictor and actual hardware predictor

Branch misses from Intel i5-5200U with Broadwell Architecture

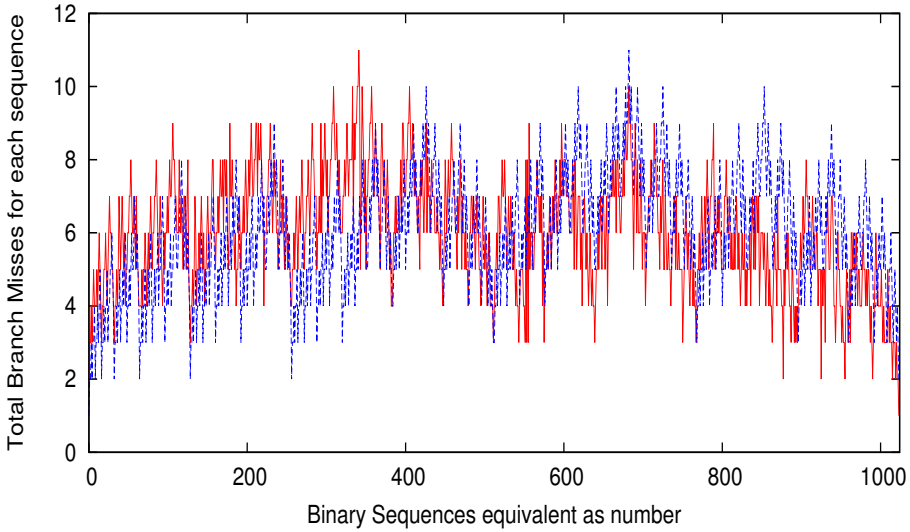


Figure: Branch misses from 2-bit predictor and actual hardware predictor

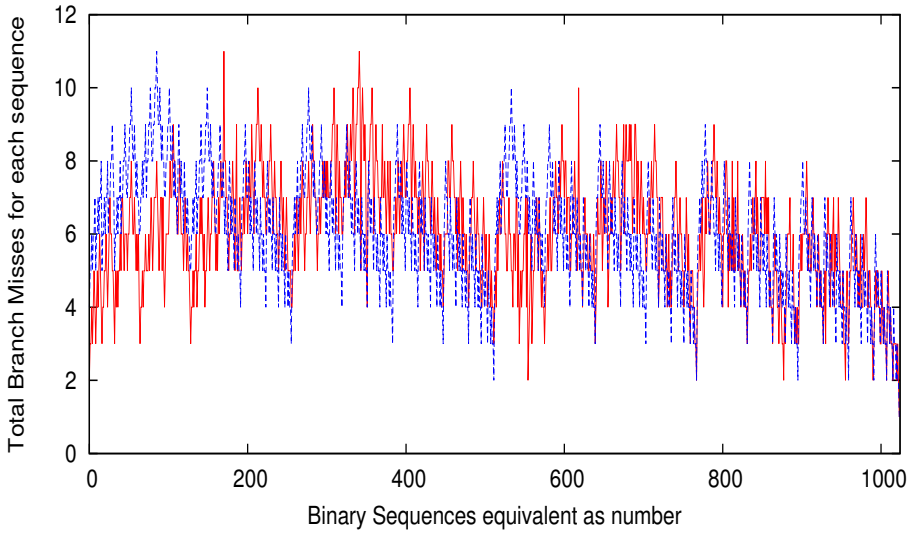


Figure: Branch misses from 3-bit predictor and actual hardware predictor

Branch misses from Intel i3-M350 with Sandybridge Architecture

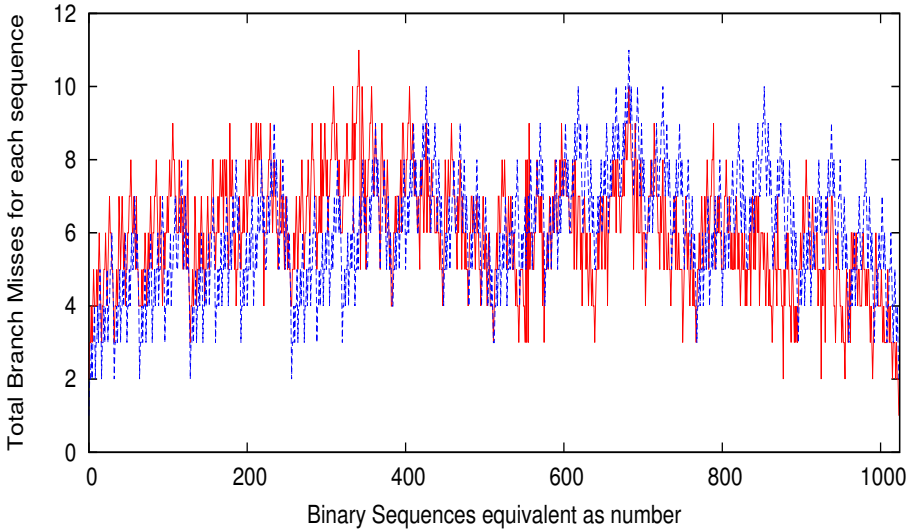


Figure: Branch misses from 2-bit predictor and actual hardware predictor

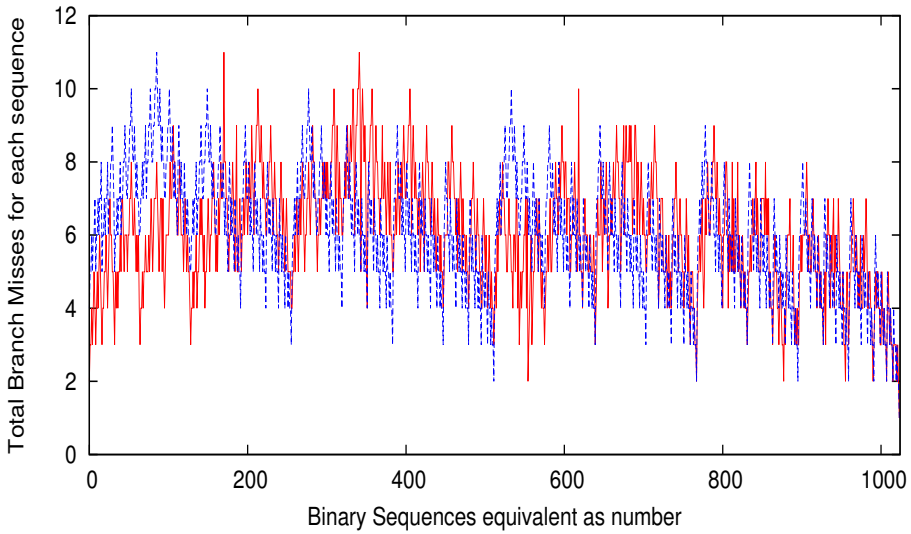
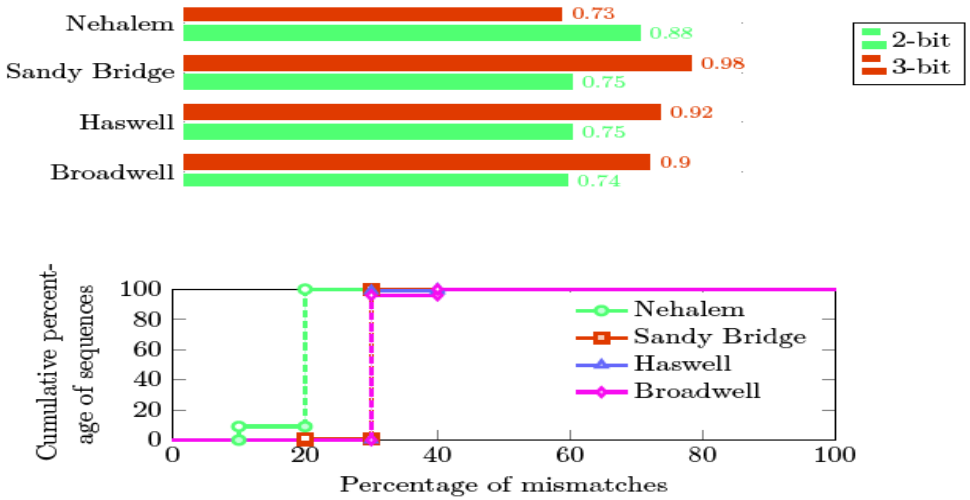


Figure: Branch misses from 3-bit predictor and actual hardware predictor

Branch misses from Intel i3-M350 with Sandybridge Architecture



(a) Predictor model: 2-bit saturating counter state machine.

(b) Predictor model: 3-bit saturating counter state machine.

Threat Model

1. The measurements made are such that an event(branch misses) is getting monitored on specific sample period of the instruction count.
2. This measurement is handled by `ioctl` interface and observed to be asynchronous in nature.
3. The attack model is both practical and realistic in shared server environment where the hardware is shared between multiple users processes.
4. In such setting, the branch mispredictions can be observed over a target execution by the attacker from another concurrently running process.

Algorithm 3: Right to left always double-add scalar multiplication algorithm.

Input: $P \in G$ and $k = (k_{t-1}, ..., k_0)_2 \in N$
Output: $Q = kP$
begin
  $R_0 = O$; $R_1 = P$
  forall $j \in \{0, 1, \ldots, t-1\}$ **do**
    $b = 1 - k_j$; $R_b = 2R_b + R_{k_j}$
  return R_0

A point addition or doubling step for Edward-1174 curve

$$b = 1 - k_j \quad R_b = 2R_b + R_{k_j}$$

$$R_1 = Z_1 \cdot Z_2,\ R_2 = {R_1}^2$$
$$R_3 = X_1 \cdot X_2,\ R_4 = Y_1 \cdot Y_2$$
$$R_5 = d \cdot R_3 \cdot R_4,\ R_6 = R_2 - R_5,\ R_7 = R_2 + R_5$$
$$X_3 = R_1 \cdot R_6 \cdot ((X_1 + Y_1) \cdot (X_2 + Y_2) - R_3 - R_4)$$
$$Y_3 = R_1 \cdot R_7 \cdot (R_4 - R_3)$$
$$Z_3 = R_6 \cdot R_7$$

Overview of the attack

We follow by a strategy of *Deduce & Remove* to target the scalar splitting and scalar blinding countermeasures on ECC.

1. n-bit scalar K.
2. m branch miss samples from the execution over K.
3. Each branch miss sample is reported after sample period of I instructions.
4. Thus effectively, each sample of reported branch misses is affected by n/m bits of the scalar K.
5. In our experiments, we have chosen I such that $n/m = 2$. Moreover, considering a b-bit predictor, I should be such that $n/m \leq b$ ($b = 3$ for our case).

Template Building for b bit predictor

Algorithm 1: Template Building

Input: n: number of bits of scalar K, m: number of branch miss samples for n bits scalar. Acquired samples of 2^t sequences iterated over i times for each of the 2^b predictor states

Output: Templates corresponding to 2^t sequences of $t * m/n$ sample points for each 2^b state

```
begin
    for 2^b states of the b-bit predictor do
        for each of the 2^t sequences do
            Construct separate distributions of i values for each t * m/n sample points.
            Compute separately, the mode of each distribution to be the elected template
            for that sample point.
        end
    end
end
```

Offline template matching for an unknown trace

- Sample trace collected for an unknown secret scalar is matched iteratively to the previously constructed templates.
- The matching phase is composed of: *Deduce* and *Remove* steps.

Deduce

- In *Deduce* phase, we start matching from the Least Significant Bit (LSB) of the scalar multiplication.
- The matching can be done iteratively taking on a trace with s sample points ($s = t * m/n$).
- These s samples are point-wise matched with all the template points for each particular template and the distances for each of the traces are measured using the Least Square Method (LSM).

Remove

- For real experiments noise is predominant, several templates may return same least square distance.
- The noise filtering is done in *Remove* step.
- Parameters chosen for *Remove* step is device-specific and can also change with algorithm.

Template matching for Exponent Scalar Splitting

1. *Acquire*: N pairs of split scalar multiplications over $K - r$ and r are acquired over t bits, each pair for unknown and random values of r.
2. *Deduce*: For each of the N pairs, corresponding pairwise template matching is performed, on each sample. It results in N values each for $K - r$ and r. Pairwise adding up of each pair $(K - r + r)$ results in t-bits of K.
3. *Remove*: Ideally, all N values of K obtained previously must be identical. The non-matching values can be removed by majority voting.

Template matching for scalar blinding

1. *Acquire*: N blinded scalar multiplication over $(K + r\#E)P$, for random, unknown values of r.
2. *Deduce*: For each of the N trace, pointwise matching over s branch misprediction samples of t bits is performed. It results N candidates for t bits for of $K + r\#E$.

3 Remove:

- Choose any 3 branch misprediction traces out of N traces, for random r_1, r_2, r_3.
- Deduce step reveals t bits of $K + r_1\#E$, $K + r_2\#E$ and $K + r_3\#E$ respectively.
- Take pair wise difference of the candidate values example $(K + r_1\#E) - (K + r_2\#E)$.
- Compute $r_1\#E - r_2\#E$, $r_2\#E - r_3\#E$ and $r_1\#E - r_3\#E$. Now for correct t bits of the blinded scalar, adding up of candidate value of $r_1\#E - r_2\#E$ and $r_2\#E - r_3\#E$ would result in non-empty set on intersection with candidate of $r_1\#E - r_3\#E$.
- Combination for empty set for intersection can be discarded, leading to t bits of blinded scalar.

Iteratively repeating this process leads to retrieval of $k + r_i\#E$ separately.

Modulus with the order of curve returns secret K for each r_i's.

Building Branch Misprediction Templates Using perf ioctl Sampling

Success of our attack is highly dependent on how accurate the template has been built.

- The templates constructed using mean value loose their correlation to the behavior of the 3-bit predictor.
- We separately construct frequency distributions for each of these sample points and select the modal value as the candidate template point.
- Templates constructed taking the highest frequency points capture the essence of the distribution accurately.

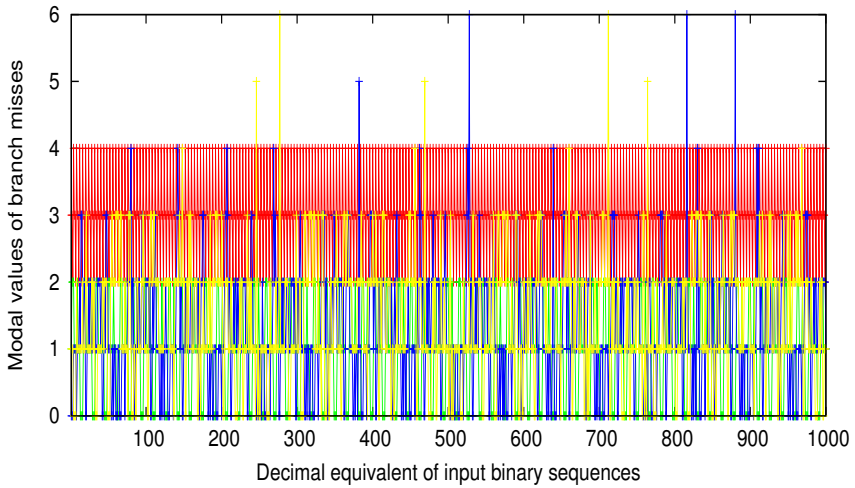


Figure: Variation of sample points of branch miss templates

Template for LSB

- The most noisy sample point is the first one, which is supposed to be affected only by the Least Significant bits.
- This sample gets affected by the branch misses from instruction before the scalar multiplication starts to execute
- We have performed a frequency analysis on the first branch miss samples observed over a set of random binary sequences of input depending on the actual values of the LSB and the bit following LSB.

Retrieving the Least Significant Bit for Scalar Splitting

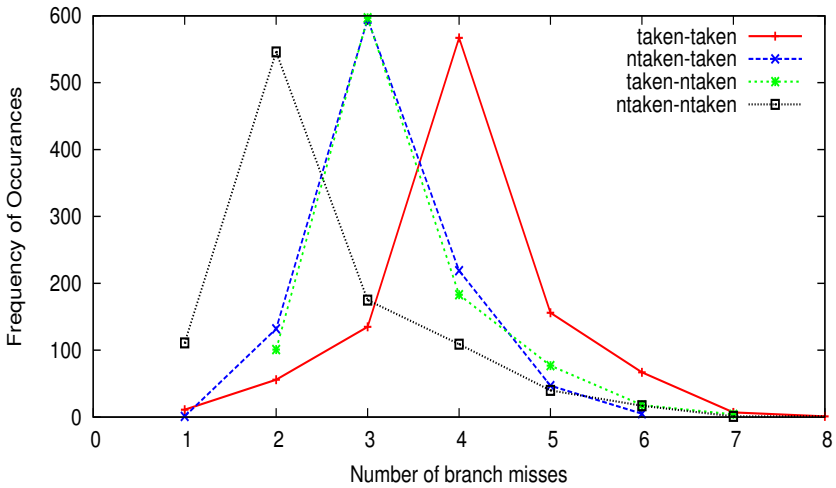


Figure: Confusion in determining the LSB for scalar splitting

- For each of the N sample pairs, we select the pairs where neither of the sample points exhibits a value 3 or 4.
- If a sample point exhibits value < 2, we classify both the branches as `not-taken` ie, the bits to be 11.
- If a sample point exhibits value 2, we conclude that the branches are either both `not-taken` or `not-taken` followed by a `taken` branch. Thus the bit values are 11 or 01.
- If a sample point exhibits value 5, we conclude that the branches are either both `taken` or `taken` followed by a `not-taken` branch. Thus the bit values are 00 or 10.
- If a sample point exhibits value > 5, we classify both the branches as `taken` ie, the bits to be 00.

Iterative Template matching for scalar splitting

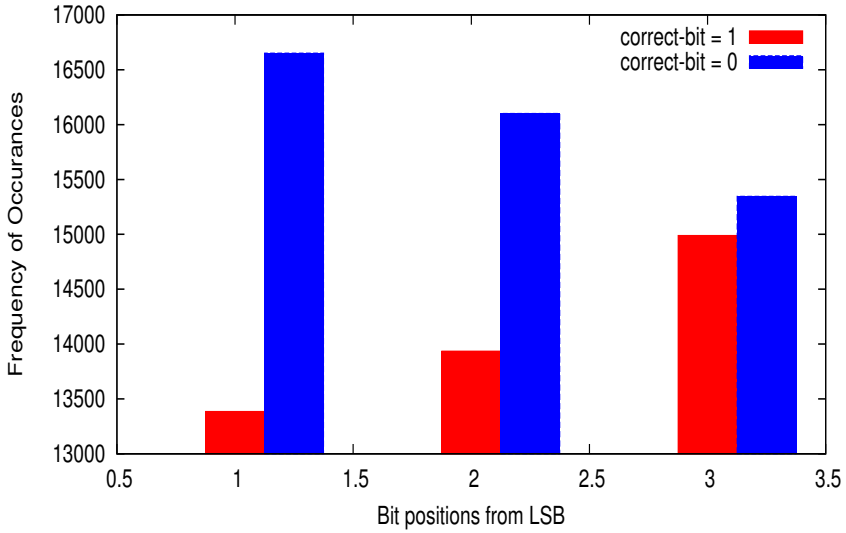


Figure: Determining next three bits

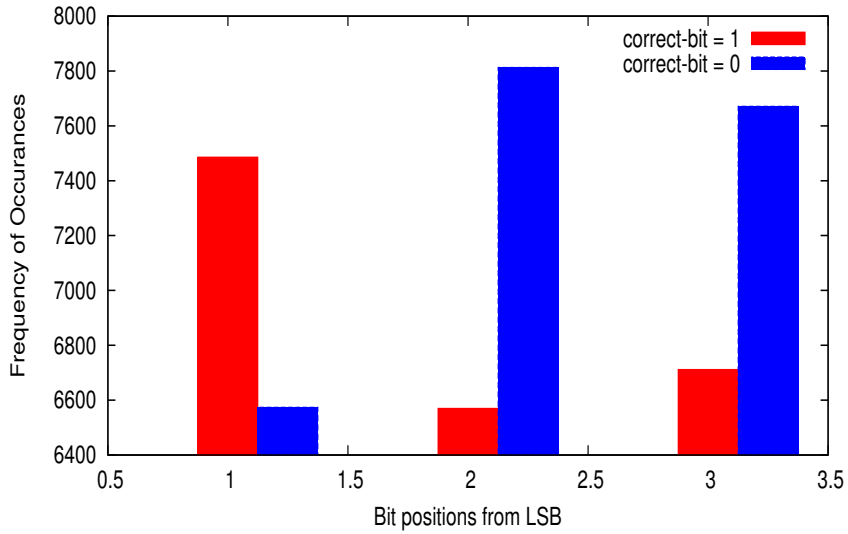


Figure: Determining further three bits

Determining 3 bits at a time for scalar splitting

Efficiency of Deduce and Remove strategy on Scalar Blinding

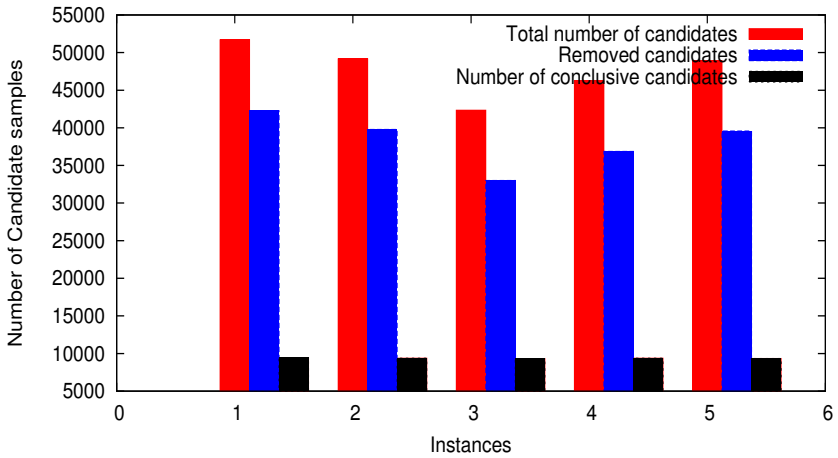


Figure: Efficiency of Deduce and Remove step on scalar blinding

- Bars with red represent the total number of candidates which were matched after template matching.
- Bar with blue indicates the total number of candidate values that were pruned.
- Bar in black represents the number of correctly retrieved candidates after taking intersection. The number of correctly retrieved ones are higher than 93%.
- Knowing 3 bits we update the state of the predictor and perform template matching on the next t bits to retrieve the following 3 bits.

Revealing Secret of Secret Splitting and Randomization

Building template for data-dependent branches in Curve1174 requires two sets of branch misprediction traces:

- the simulated mispredicted traces from 3-bit predictor for each modular reduction operations involved in a point addition operation,
- the perf samples corresponding to the same set of inputs.

The i^{th} bit of the blinded scalar is guessed and both 0 and 1 is appended to the already known $(i-1)$ bits for each of the j blinded scalar sequences. $bm\_perf_{(i=0),j}$, $bm\_perf_{(i=1),j}$ as branch misprediction samples for only the guessed bit, where the operation is performed over all j sequences and known $(i-1)$ bits for two separate guesses. Thus we have $2 \cdot j$ sequences and we separately take the `ioctl` branching samples where the i^{th} bit is guessed to be 0 and 1 respectively.

Similarly, we have the simulated branch misses from the 3-bit predictor $bm\_sim^{0}_{i,j}$ and $bm\_sim^{1}_{i,j}$ for both guesses as discussed earlier. The steps to build a template are as follows:

- For each k instances of the modular reduction operation, we apply a windowing technique to $bm\_perf_{(i=0),j}$ and $bm\_perf_{(i=1),j}$ to identify approximately how many samples are responsible for each modular reduction operation.
- Now for $guess = 0$, we consider $bm\_perf_{(i=0),j}$ and $bm\_sim^{0}_{i,j}$, and separately build template points based on whether or not they suffer from a branch miss.
- For $guess = 0$, we separately consider templates from the samples in $bm\_perf_{(i=0),j}$ for each of the k modular reduction operations involved for point addition for each of the j sequences.

We construct two bins for each of the k modular reduction operations based on whether they have a simulated branch miss as listed in $bm\_sim^{0}_{i,j}$.

Let us consider a particular modular reduction in k, we perform a vertical analysis of all sequences in j. We separate the sequences into two bins based on whether they have a simulated branch miss at particular modular reduction step in $bm\_sim^{0}_{i,j}$. Now we fill the $bin\_miss_{(i=0),k}$ with samples from $bm\_perf_{(i=0),j}$ for the k^{th} particular modular reduction, if there has been a simulated branch miss in $bm\_sim^{0}_{i,j}$. Otherwise, we fill $bin\_no\_miss_{(i=0),k}$ if there is no misprediction.We separately construct templates taking the mode of the distributions of each of these constructed bins as described in the previous discussions of template building.

At the end of this step, we have $2 \cdot k$ separate bins for all the k modular reduction operations considering all the j sequences where all the i^{th} bit has been guessed to zero. A similar construction can be performed with i^{th} bit being guessed as 1.

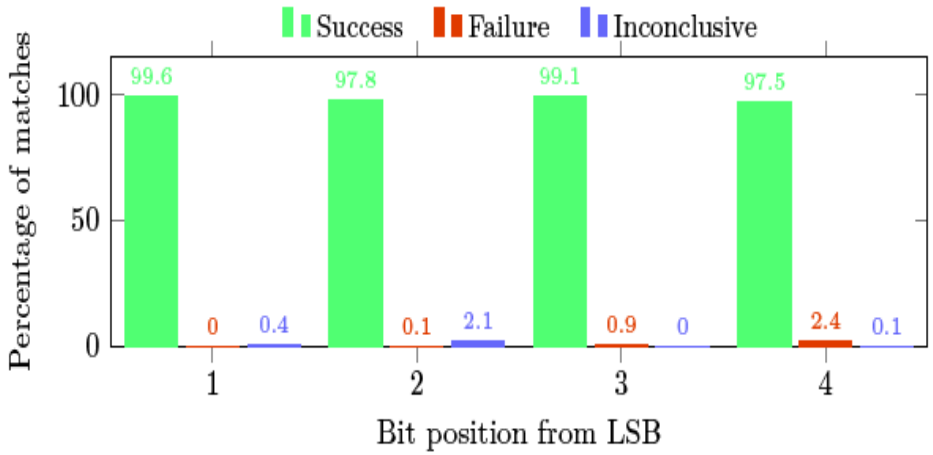


Figure 6: Probabilities of bit retrieval for scalar splitting in Curve1174.

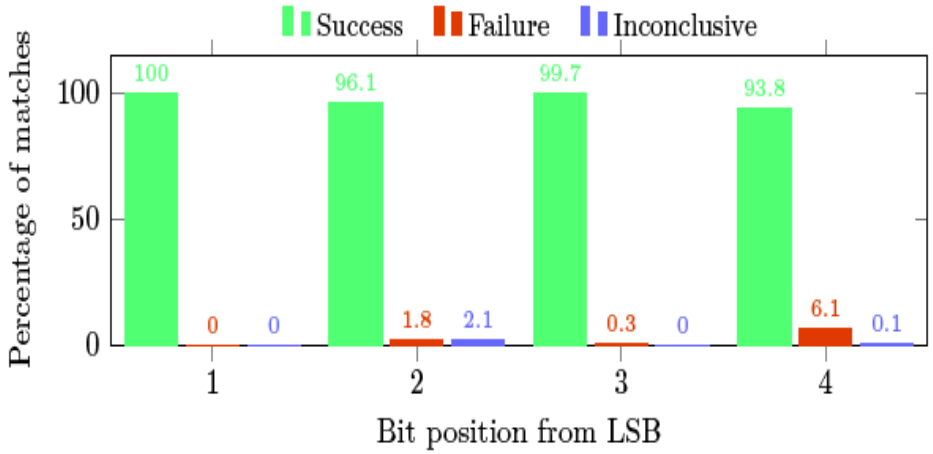


Figure 7: Probabilities of bit retrieval for scalar randomization in Curve1174.

Countermeasures

1. The most obvious countermeasure is to avoid conditional if-else implementaton.
2. Another countermeasure to thwart such attacks is to randomize the state of the predictor intermediate to the execution.
3. But this does not ensure complete security, since the adversary can be more powerful having more granular traces and even then these countermeasures will pose themselves ineffective.

Summary

- We initially perform reverse engineering on the branch predictor hardware of Intel's Broadwell and Sandybridge systems and show that the hardware has a significant similarity in behavior to the 3-bit predictor algorithm.
- Subsequently, we use this granular observation of branch misprediction to attack the DPA secure implementations of ECC.
- The experimental results illustrate the effectiveness and efficiency of our proposed attack for both scalar splitting and scalar blinding countermeasures.

Thank You