Who watches the watchmen? : Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms

Sarani Bhattacharya[1] and Debdeep Mukhopadhyay[1]

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur–721302, India
Contact E-mail: {sarani.bhattacharya, debdeep}@cse.iitkgp.ernet.in

Abstract. Asymmetric-key cryptographic algorithms when implemented on systems with branch predictors, are subjected to side-channel attacks exploiting the deterministic branch predictor behavior due to their key-dependent input sequences. We show that branch predictors can also leak information through the hardware performance monitors which are accessible by an adversary at the user-privilege level. This paper presents an iterative attack which target the key-bits of 1024 bit RSA, where in offline phase, the system's underlying branch predictor is approximated by a theoretical predictor in literature. Subsimulations are performed to classify the message-space into distinct partitions based on the event branch misprediction and the target key bit value. In online phase, we ascertain the secret key bit using branch mispredictions obtained from the hardware performance monitors which reflect the information of branch miss due to the underlying predictor hardware. We theoretically prove that the probability of success of the attack is equivalent to the accurate modelling of the theoretical predictor to the underlying system predictor. Experimentations reveal that the success-rate increases with message-count and reaches such a significant value so as to consider side-channel from the performance counters as a real threat to RSA-like ciphers due to the underlying branch predictors and needs to be considered for developing secured-systems.

Keywords: Branch misprediction, HPC, public-key cipher, side-channel.

1 Introduction

Micro-architectural features leave footprints in the processor which is often captured by side-channels. Side-channel attacks allow malicious user to gain access to sensitive data of the system under attack by monitoring power consumption, timing, or electro-magnetic radiation of the microprocessor. In recent microprocessors, various architectural components are incorporated in the system to improve the system performance and these are emerging as new sources of side-channel leakage.

In the pioneering work in [1] it was first shown that the time to process different inputs can be used as a side-channel information to find the exponent bits of the secret keys for RSA, Diffie-Hellman, DSS etc. In [2], the penalty for mispredicted branches in number of clock cycles is observed as side-channel to

identify the data dependent operations of the public-key cryptosystem. On a standard RSA implementation, four different types of attacks were performed exploiting the Branch Prediction Unit (BPU) by using both synchronous and asynchronous techniques. Using timing as the side-channel in [2], the misprediction information is modeled to identify the secret key. While in the synchronous and asynchronous attacks the Branch Target Buffer (BTB) is modified by the attacker to surface the attack.

Hardware performance counters (HPCs) are a set of special- purpose registers to store the counts of hardware-related activities within the microprocessor. These counters contain rich source of information of the internal activities of the processor and hence can find usage for both attacks and their countermeasures. In [3], these HPCs are exploited as side-channels for time based cache attacks. HPC L1 and L2 D-cache miss counters have been exploited as side-channels in [3] for performing timing based cache attacks on symmetric-key algorithms, like AES. While the paper shows that the HPCs can be used as potential source of leakage, the attacks were sensitive to noise introduced through loops, branches and also compiler optimizations to retain the tables. In this paper, we show that asymmetric ciphers like RSA, which does not have tables and have several branches and due to the underlying algorithm and the internal multipliers used, can be successfully attacked by monitoring the event branch miss through HPCs.

In this attack, we target the branch-predictors which were previously shown to lead to attacks using timing as side-channel [2]. Several research work has been developed to thwart these attacks by fuzzying the timestamp counters, adding noise etc. However, we show that powerful side-channels may still exist through the HPCs which monitor the branch misses at the user-privilege. Interestingly, we show through real experiments that though the underlying branch predictors are unknown, the attacker can approximate them by theoretical models which correlate well with the actual statistics of branch misses. Using these approximations, one can launch an attack and successfully recover a full 1024-bits key of RSA algorithm implemented with key bit dependent conditional operations. The modular exponentiation of RSA has been implemented using both naïve square and multiply and Montgomery ladder, while the underlying multiplication and squaring has been implemented using Montgomery's method. The attack iteratively recovers the key bits and has two distinct phases:

- An offline phase, during which the system branch predictor is approximated by a theoretical model (namely, two-bit, two-level adaptive predictor) and is used to classify the message space into distinct partitions based on the event of branch misprediction and assuming the value of the i^{th} key bit.
- In the online phase, we perform the actual attack to ascertain the i^{th} key bit using the branch mispredictions obtained from the values of the performance monitors which provides us with the real information of the branch miss due to the actual predictor hardware in the architecture.

We provide a theoretical proof to justify that the probability of success is equivalent to how closely the theoretical predictor models the underlying system predictor hardware to guess the i^{th} bit correctly. It is also noted that success probability increases with number of messages and reaches a significant value to

consider the side-channels due to performance counters a real threat to RSA-like ciphers exploiting the underlying branch predictors. What makes this result more relevant is the fact that protections which fuzz the timing channels are not sufficient to thwart these attacks, and presents performance counters as a distinct side-channel which needs to be considered for developing secured systems.

In the later part of this paper, we extend our attack to the RSA-OAEP randomized padding procedure where we target the decryption phase of the implementation and the branch miss side-channel information of the entire decoding procedure can be successfully exploited to reveal the secret exponent.

The organization of the paper is as follows:- The following Section 2 provides a brief idea on modular exponentiation algorithms and some well-known predictor algorithms. In Section 3 we demonstrate the vulnerability due to the event "branch-misses" as side-channel. The attack algorithm is described in Section 4 with the detailed analysis on the retrieval of secret key bits in two phases. A formal analysis on the success of the algorithm is presented in Section 5. Section 6 provides the experimental validation for the attack strategy. A brief discussion on the future prospects of the work and some probable countermeasures are provided in Section 7 and the final section contains conclusion of the work we present here.

2 Preliminaries

In this section, we provide a background on some key-concepts, which include some implementation algorithms for public-key ciphers and some well-known branch predictors which have been subjected to attack.

2.1 Exponentiation Algorithms and Underlying Multiplication Primitive

In RSA-like asymmetric-key cryptographic algorithms, inputs(M) are encrypted and decrypted by performing modular exponentiation with modulus N on public or private keys represented as n bit binary string. While during encryption the exponent(e) is public, the target for attackers is the exponentiation carried out while decryption, where the secret key(d) is used as the exponent. The most popular algorithm to implement modular exponentiation is the square and multiply algorithm. The square and multiply algorithm as described in Algorithm 1, performs squaring at each step, while there is a conditional multiplication operation which is performed only if the exponent bits are set. This algorithm performs unbalanced instruction execution because multiplication statements are conditioned on the exponent bit.

Due to this extra computation step(which is being conditioned on the secret exponent bit), simple power attacks (SPA) and timing attacks exploit this conditional instruction execution and eventually retrieves the secret exponent.

A naïve modification to protect the side-channel leakage of square and multiply exponentiation algorithm is to have a balanced instruction execution and is proposed in the Montgomery ladder algorithm [4] explained in Algorithm 2.

Algorithm 1: Binary version of Square and Multiply Exponentiation Algorithm

```
begin
    S ← M ;
    for i from 1 to n − 1 do
        S ← S ∗ S mod N ;
        if d_i = 1 then
            S ← S ∗ M mod N ;
        end
    end
    return S ;
end
```

Algorithm 2: Montgomery Ladder Algorithm

```
begin
    R_0 ← 1
    R_1 ← M
    for i from 0 to n − 1 do
        if d_i = 0 then
            R_1 ← (R_0 ∗ R_1) mod N
            R_0 ← (R_0 ∗ R_0) mod N
        end
        else if d_i = 1 then
            R_0 ← (R_0 ∗ R_1) mod N
            R_1 ← (R_1 ∗ R_1) mod N
        end
    end
    return R_0
end
```

This algorithm performs the entire exponentiation by alternatively modifying the values of two dummy variables depending on the exponent bits. Algorithm 2 has both "if" and "else" statements, and everytime one of the two possible sets of instructions are getting executed. Unlike the square and multiply algorithm, here the number of squarings and multiplications executed will always be constant and equal to the length of the key which inhibits simple power attack and timing attack.

A highly efficient algorithm for performing modular squaring and modular multiplication operation (in these modular exponentiation algorithms) is the Montgomery Multiplication Algorithm [5], since it avoids the time consuming integer division operation. Montgomery Multiplication as in Algorithm 3 computes modular multiplication of form $a * b(\text{mod}N)$. If the RSA modulus N is a k-bit number then a variable R is assumed to be 2^k. Montgomery Multiplication calculates $Z = A * B * R^{-1}(\text{mod}N)$ where $A = a * R(\text{mod}N)$, $B = b * R(\text{mod}N)$ and $R^{-1} * R = 1(\text{mod}N)$. There is an extra reduction step at the 4^{th} line of the Algorithm 3 which is particularly of interest to the attackers. The conditional execution of the reduction statement depend on the inputs, thus can be exploited in modular exponentiation scenario to surface a timing attack.

In situations when both public key exponent e and input m are small then the modular exponentiation can be reverted efficiently and the encryption fails to fulfill the criteria for asymmetric key ciphers. RSA being a deterministic algorithm is not semantically secure and an intelligent adversary can launch known ciphertext attacks on this cipher. This effectively leads to message padding schemes which encodes messages first then allows encryption on these encoded messages.

Algorithm 3: Montgomery Multiplication Algorithm

```
begin
    S ← A * B ;
    S ← (S + (S * N^{-1} mod R) * N)/R ;
    if S > N then
        S ← S − N ;
    end
    return S ;
end
```

In the next subsection a brief overview of randomized message padding procedure is provided.

2.2 RSA-OAEP Randomized Padding Scheme

RSA encryption along with PKCS#1 v1.5 encoding was shown to be insecure in [6] as it reveals information regarding the plaintexts by examining the ciphertext in polynomial time. To overcome this security problems of the chosen ciphertext attacks, OAEP encoding scheme was introduced to detect any manipulation while decrypting the ciphertext and outputs an error message if any tampering with the ciphertext is performed.

In RSA-OAEP randomized padding procedure, the public key encryption(as in modular exponentiation) is performed on the encoded message (which we refer to as the plaintext) instead of the original message(though in the previously stated algorithms the plaintext is same as the message). The decryption and decoding procedure in RSA-OAEP is reverse to the encryption process and is illustrated in Figure 9. The input ciphertext is decrypted with the secret key to reveal the plaintext corresponding to the ciphertext. The plaintext while decoded as in Figure 9 refuses to output any message if the specifications of the decrypted ciphertext string is not met. The criteria are illustrated in diagonal boxes in the Figure 9 and if violated, the decoding process outputs "error message".

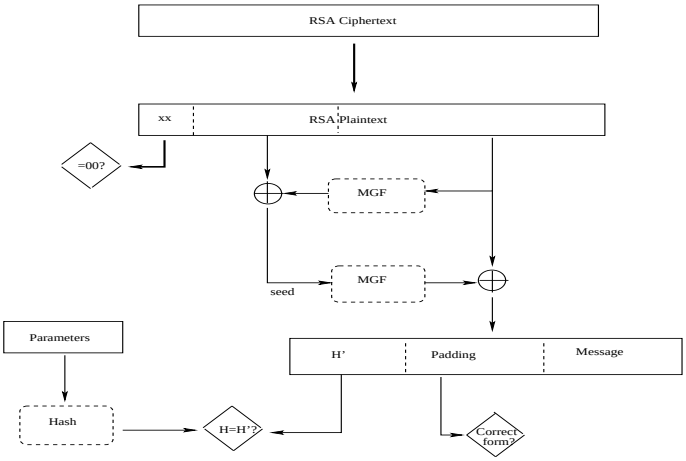


Fig. 1. Decryption in RSA-OAEP procedure [7]

The detailed specifications to the Mask Generation Function(MGF), hash function, selection of parameter and seed generation are provided in [8]. The existing side-channel attacks against this scheme exploits fault and timing analysis on three checking conditions separately to identify the ciphertexts(which can be decoded to messages successfully) in Chosen-ciphertext attacks.

This paper evaluates the security of implementations of public-key ciphers on standard processors using branch misses from the hardware performance counters(HPCs). The leakage is caused due to the presence of branch predictors in the modern architecture. Some of the very popular branch predictor algorithms are explained in the next subsection.

2.3 Dynamic Branch Predictor

The 2-bit dynamic branch predictor state machine is one of the various predictor algorithms that is most oftenly used in practice [9]. This is a deterministic algorithm predicting next branch to be *taken* or *not taken* depending on the history of previously taken branches. In a 2-bit prediction scheme the predictor must miss twice before the prediction changes. But conditional branches that are taken in a regular recurring pattern are not predicted well by this bimodal predictor.

In such cases a two-level adaptive predictor [10] works better as the predictor remembers the last k occurrences of a branch instruction and uses a s-bit prediction function (such as a s-bit predictor) for each of the 2^k history of patterns. The first level of the two level adaptive predictor uses a *branch history register*, which is a shift register storing the history of the last k branches. The branch history register indices to a second level called *pattern history table*, which can hold 2^k entries, each s bits wide. When a conditional branch say B is getting predicted, content of the k bit history register is used as address to the pattern history table. Let S_c be the contents of the pattern history table. S_c is fed to the s-bit prediction decision function (such as the 2-bit dynamic predictor), which outputs the predicted value $\lambda(S_c)$. The state machine of both of this predictors are provided in Appendix A.

In the next section we will provide a brief motivation for considering branch misses from performance counters as side-channel to attack public-key ciphers.

3 Modelling Branch Miss as Side-Channel from HPC

3.1 Using event Branch-misses as Side-channel

In this work, hardware performance counters(HPCs) are exploited to monitor side-channel information of the **number of branch misses** on the **square and multiply** exponentiation algorithm which uses Montgomery multiplication algorithm as subroutine for the operations like squaring and multiplication. As observed in Algorithm 1, the code while in execution can proceed in any of two paths, since the multiplication operation is performed only if the particular exponent bit is set. In addition to this, the Montgomery multiplication subroutine used for the squaring and multiplication operation also has an extra conditional reduction statement which gets executed when the intermediate input exceeds

the modulus N. Thus, there exists a side-channel information via the hardware performance event **"branch-misses"**. Though timing side-channel can also be used to monitor the misprediction delays due to branch misses but when we wish to exploit only the branch mispredictions, measuring the time of a misprediction delay (of an event when measured from a multitasking system) is less significant compared to actually monitoring the event branch misses through HPCs.

The side-channel leakage through branch miss is caused due to the presence of underlying branch predictor in architecture. Branch misses rely on the ability of the branch predictor to correctly predict future branches to be taken. If the prediction is false, the instruction pipeline is flushed leading to a branch miss. Thus the branch predictors play a major role in correctly predicting the next target instruction and reducing the misprediction penalty.

The performance counters leak information of branch misses while exponentiation operation is performed on the secret exponent bits for the public-key ciphers. The profiling of the HPCs can be done using performance monitoring tools and is considered as a side-channel source since it provides a simple user interface to different hardware event counts.

3.2 Strong correlation between two-bit predictor and system branch predictor

State machine of the 2-bit dynamic predictor as explained in Section 2.3 has been extensively used as an underlying predictor in the older versions of the Intel family of microprocessors [11]. But the actual predictor structure in architecture(inbuilt in the recent processors) is not disclosed by the processor manufacturers. In order to monitor the information of branch misses from the HPCs, we aim to exploit a strong correlation of branch misses from the actual inbuilt predictor and some of the well-known predictor algorithms. In order to approximate the branch mispredictions from system's underlying predictor algorithm, we first made an installation of Perf tool on Linux OS Ubuntu 12.04.1 LTS to monitor the event "branch-misses", which indicates number of branch mispredictions suffered by an executable. The following **command can be executed at the user privilege**.

$ perf stat -e branch-misses executable-name

With the aim of approximating the underlying system predictor with the well known 2-bit dynamic predictor, branch misses for performing exponentiation are observed on 10000 separate random keystream, each of 1024 bits on Intel i5 platform. An observation on the number of branch misses simulated from the 2-bit dynamic predictor and the corresponding branch misses as obtained through performance counter values is illustrated in Figure 2. Two sets of information are correlated in the following manner

- Each of these 1024 bit random key is simulated on 2-bit dynamic predictor and the number of branch misses are observed for each of them.
- The number of branch misses are also observed from the performance monitoring tool over the square and multiply exponentiation algorithm for each of the random keystreams. The branch miss information for a particular key is averaged after exponentiations over 1000 inputs to reduce noise.

– The number of branch misses obtained from performance counters is found to be increasing as the total number of predicted branch misses on a key-stream increases as in Figure 2.

The absolute values of branch misses obtained from HPCs as plotted in Figure 2 are much larger than the theoretically simulated values of the 2-bit predictor algorithm. It may appear to the observer to be counter-intuitive since the actual branch predictors in hardware are much sophisticated compared to the primitive 2-bit dynamic predictor. But the rationale behind this may be explained as that the HPCs report branch miss statistics for the entire execution of the executable and thus are affected by the environmental running processes as well.

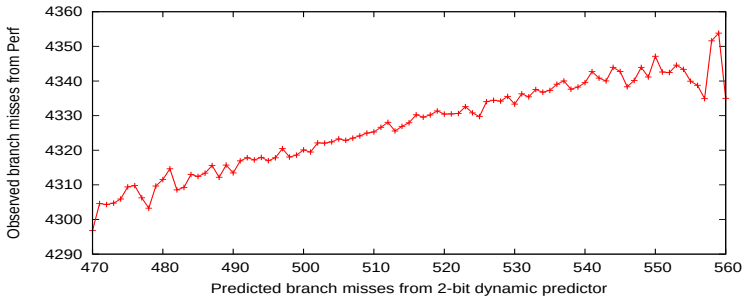


Fig. 2. Variation of branch-misses from performance counters with increase in branch miss from 2-bit predictor algorithm

A direct correlation is observed in Figure 2 for the branch misses from performance counters and branch misses from the simulated 2-bit dynamic predictor over a sample of exponent bitstream. This confirms our assumption of 2-bit dynamic predictor being an approximation to the underlying system branch predictor and in our work, we modelled this strong effect of the bimodal predictor to exploit the side-channel leakage of branch misses from the performance counters. As a further extension, we also perform the attack by approximating the branch predictor by a two level adaptive predictor, where the second level is a dynamic 2-bit predictor model itself. We later show that the accuracy of our attack improves with the correlation between the actual and the model assumed, which is quite high as also supported by our experiments.

4 Attack Algorithm featuring Performance counters monitoring branch misses

In the attack algorithm, we claim to identify the secret bit by utilizing the behavior of the well known predictor algorithms as an approximation to underlying system branch predictor, to simulate the mispredictions for initial known secret exponent bits over each input ciphertexts. Later we perform an analysis phase based on branch misprediction information from actual HPCs to reveal secret bits. The attack is an adaptation of direct timing attack demonstrated in [2],

where the paper talks about observing a separation in timing between distinct input sets, the sets being separated by a hypothetical predictor algorithm. The hypothetical attack scenario presented in [2] cannot be implemented on real systems until and unless the adversary gets to know the actual structure of the branch predictor architecture of the target system. None of the leading processor manufacturers publish their architectural details since this puts their intellectual property at risk, making the whole idea of proposed attack unrealistic. In this present work, we extend the attack algorithm and the novelty of the work lies in the fact that the 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. In order to target real systems, we perform the subsimulations on some well-known predictors like 2-bit dynamic predictor and two-level adaptive predictor as they approximate the real predictor to a great extent in order to partition the entire ciphertext set into smaller ones. In the latter phase, we perform actual experiments using branch misses from HPCs as side-channel to ascertain the secret bit.

4.1 Threat Model for the attack

The basic assumptions of the attack is that the adversary targets the modular exponentiation while RSA decryption is taking place. The attacker knows the first i bits of the private key and he wants to determine next unknown bit d_i of the key $(d_0, d_1, \cdots, d_i, \cdots, d_{n-1})$. The attack algorithm runs in two phases, where in the offline phase for an input m, the attacker can only simulate for the partially known bits and the assumed target bit. In this phase, the subsimulations for each input is fed to a predictor model to generate mispredictions and based on this event misprediction, the entire ciphertext set is partitioned. The adversary neither has an access to the HPCs nor any access to do a partial computation on the target machine. Whereas in the online phase, the adversary can only observe branch misses over entire secret key for various input ciphertexts. In this phase, the attacker is not allowed to perform any subsimulation on the secret key.

In the next subsection, we present an iterative attack algorithm in two phases where the following analysis can be performed to identify individual secret key bits one after another.

4.2 Offline Phase

In this phase, the adversary partitions a sample input set M by simulating the branch mispredictions for the conditional reduction of Montgomery multiplication at the $(i+1)^{th}$ squaring step of Square and multiply algorithm. For any input $m \in M$, the attacker can simulate the execution of the exponentiation algorithm for the initial i bits (that are already known) and can generate a trace of branches as $(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$ following steps of Algorithm 1, 3. Here $t_{m,i}$ is simulated as either a taken or not taken branch depending whether the conditional reduction branch statement at the i^{th} squaring operation is being executed. As we already have the knowledge of bits $(d_0, d_1, \cdots, d_{i-1})$, the trace of branches can be simulated by the attacker as $(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$.

At this stage, the adversary assumes both $d_i = 0$ and 1, and separately does the following analysis in the offline phase. Under the assumption of d_i having value j, where $j \in \{0, 1\}$, appropriate value of $t^j_{m,i+1}$ is simulated. This situation is illustrated in Figure 3. The $(i+1)^{th}$ squaring (being executed by Montgomery multiplication subroutine), the execution of an extra reduction step at line 4 (of Montgomery Multiplication as in Algorithm 3) is purely dependent on the sample input m as well as value of unknown d_i.

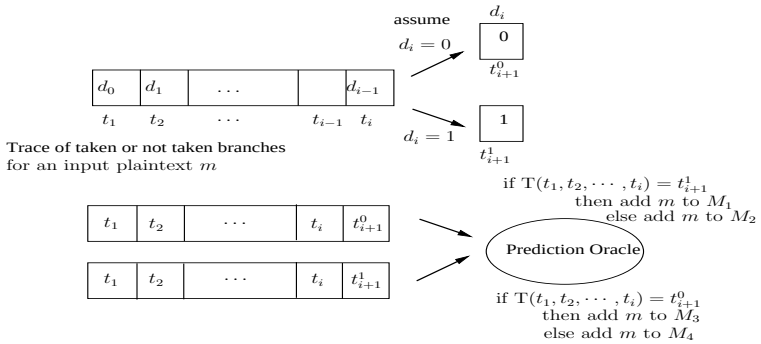


Fig. 3. Partitioning randomly generated Ciphertexts set based on simulated Branch miss Modelling

For the simulated branch history traces for a random ciphertext m, a misprediction occurs at $(i+1)^{th}$ squaring only if the theoretically predicted branch for $(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$ observes a mismatch with the $t^j_{m,i+1}$-th branch execution. Let the theoretical predictor be T and $(i+1)^{th}$ bit predicted by it be $p_{m,i+1} = T(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$. The partitioning of the ciphertext set is performed based on this **simulated misprediction**. The algorithm for partitioning is explained step by step in Algorithm 4 and is also illustrated in Figure 3. So the attacker can create 4 different sets due to a misprediction event during the Montgomery Multiplication(MM) at $(i+1)^{th}$ squaring:

1. $M_1 = \{m | m$ does not cause a misprediction 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 misprediction 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 $d_i = 0\}$

But there may exist a situation for a ciphertext m, such that $m \in M_2$ and $m \in M_4$, which may suffer a misprediction when d_i is assumed to be 0 as well as 1. Thus these ciphertexts may add to the noise while actually determining the secret bit, as in this case, the event misprediction does not signify whether $d_i = 0$ or 1. Likewise for the sets M_1 and M_3. Hence we ensure that there must be no common ciphertexts in the sets (M_1, M_3) and (M_2, M_4) and the sets should be disjoint. These 4 sets of ciphertexts are generated by the attacker in the offline phase.

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 $d_i = 0$, generate $t^0_{m,i+1}$;
    Similarly, assume $d_i = 1$, generate $t^1_{m,i+1}$;
    $p_{m,i+1} = T(t_{m,1}, t_{m,2}, \cdots, t_{m,i})$;
    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
  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

The Offline phase for the Montgomery Ladder algorithm differs to some extent from subsimulation and misprediction computation of square and multiply. In the Montgomery Ladder Algorithm 2, both the squaring and multiplication operations are conditioned on the exponent bits and in addition to this, there are two sets of squaring and multiplication operations that are getting executed in Algorithm 2. For a input ciphertext m, when exponent bit is 0 then lines 6,7 are executed otherwise lines 8,9 are getting executed. If we target to observe misprediction for the conditional reductions of the squaring statement, then unlike the square and multiply algorithm, the subsimulation generates two traces for taken and not taken branches (two traces correspond to squarings at line 7 and 9 respectively) for the partially known key. Similar to the previous strategy in order to identify the secret bit d_i, we assume the target bit d_i to be both 0 and 1 and separate ciphertext into 4 sets. When we assume $d_i = 0$, then mispredictions are simulated over the trace corresponding to line 7 and alternatively for line 9 when $d_i = 1$. The partitioning of ciphertexts as well as the Online phase are exactly same as explained for the square and multiply algorithm.

4.3 Online Phase

In the Online phase, branch misses from the HPCs are monitored for execution of cipher over the entire secret key for each ciphertexts in all of the 4 sets while the RSA decryption is taking place. Let the branch mispredictions observed $\forall m \in M$ from the HPCs for decryption of the cipher, forms a distribution of branch misses and we denote such distribution as $\mathcal{M}$. Branch misses for exponentiation are

monitored on each ciphertexts for these 4 separate sets M_1, M_2, M_3, M_4 for the entire secret key and results in 4 distinct distributions $\mathcal{M}_{M_1}$, and so on.

Since the i^{th} bit of the exponent can either be 0 or 1 and cannot be both at the same time, intuitively from these two pair of sets - (M_1, M_2) and (M_3, M_4), one of the pair corresponding to the correct assumption of d_i will show a consistent positive difference in the observed branch misses while in the other pair, the differences will be zero or negative. This is due to the fact, if the classification is correct, then expected mispredictions of one set (which stores the ciphertexts causing a misprediction) should be greater than the other set. If the guess is wrong, the classification being random does not exhibit this statistics.

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 $(nb_i) = 0$

5 Formally modelling the Success

In this section we claim that the success of correctly identifying the actual key bits can be alternatively stated as, how closely the theoretical dynamic 2-bit predictor follows the real predictor which is inbuilt in the processor.

In the Offline phase of the attack algorithm, for an assumption of the secret bit the set of ciphertexts M was separated in two disjoint sets based on the criteria whether they suffer from a simulated misprediction at the conditional reduction statement of $(i+1)^{th}$ squaring step. Essentially in the offline phase,

$\Pr[m_1 \in M_1] = \Pr[p_{m_1,i+1} = t^1_{m_1,i+1}]$

$\Pr[m_2 \in M_2] = \Pr[p_{m_2,i+1} \neq t^1_{m_2,i+1}]$ (assuming $d_i = 1$)

and

$\Pr[m_3 \in M_3] = \Pr[p_{m_3,i+1} = t^0_{m_3,i+1}]$

$\Pr[m_4 \in M_4] = \Pr[p_{m_4,i+1} \neq t^0_{m_4,i+1}]$ (assuming $d_i = 0$)

Also, since we remove duplicate elements from (M_1, M_3) and (M_2, M_4) in the Offline Phase, for any input m, if $m \in M_1$ then $m \notin M_3$, thus $m \in M_4$. Alternatively, we can say, $\forall m \in M, \quad t^0_{m,i+1} \neq t^1_{m,i+1}$.

While in the Online Phase, let nb_i be the bit which the attacker concludes to be the next secret bit by monitoring branch misses from HPCs for the corresponding plaintext sets following the attack algorithm. Let the expectation of the distribution of branch misses $(\mathcal{M}_M, \forall m \in M)$ be $\overline{\mathcal{M}_M}$. Thus we can decide the next bit by defining the following probabilities, for $\forall m_i \in M_i, i \in 1, 2, 3, 4$ as:

$\Pr[nb_i = 0] = \Pr[(\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) > 0 \wedge (\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) < 0]$

$\Pr[nb_i = 1] = \Pr[(\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) > 0 \wedge (\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) < 0]$ These **observed mispredictions** are actually affected by the deterministic algorithm of underlying real predictor of the system. Let us assume that the real predictor inbuilt in the system be R and $(i+1)^{th}$ bit predicted by the real predictor for the known trace is $r_{m,i+1}$ for input m. Let the $i + 1^{th}$ branch instruction has trace $B_{m,i+1}$

for unknown bit d_i. If $d_i = 0$, then $B_{m,i+1} = t^0_{m,i+1}$, otherwise if $d_i = 1$, $B_{m,i+1} = t^1_{m,i+1}$. Thus we can rewrite the previous equation as

$$\begin{aligned}\Pr[nb_i = 0] &= \Pr[(\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) > 0 \wedge (\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) < 0] \\ &= \Pr[(r_{m_4,i+1} \neq B_{m_4,i+1}) \wedge (r_{m_3,i+1} = B_{m_3,i+1}) \\ &\wedge (r_{m_2,i+1} = B_{m_2,i+1}) \wedge (r_{m_1,i+1} \neq B_{m_1,i+1})]\end{aligned} \tag{1}$$

Similarly,

$$\begin{aligned}\Pr[nb_i = 1] &= \Pr[(\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) > 0 \wedge (\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) < 0] \\ &= \Pr[(r_{m_2,i+1} \neq B_{m_2,i+1}) \wedge (r_{m_1,i+1} = B_{m_1,i+1}) \\ &\wedge (r_{m_4,i+1} = B_{m_4,i+1}) \wedge (r_{m_3,i+1} \neq B_{m_3,i+1})]\end{aligned} \tag{2}$$

Since an attacker is unaware of the underlying predictor model, the correctness of separation relies on the criteria that how closely the theoretical predictor approximates the real one. Thus the extent of correct partitioning of the random ciphertext set relies on the efficiency of the theoretical predictor model. We define the event *Success* as true if the maximum difference in branch misses is observed from the HPCs over input sets for the correct assumption. In other words,

- If difference in average branch miss $(\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) > 0$, $(\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) < 0$ and the secret bit is actually 0.
- If difference in branch miss $(\overline{\mathcal{M}_{M_2}} - \overline{\mathcal{M}_{M_1}}) > 0$, $(\overline{\mathcal{M}_{M_4}} - \overline{\mathcal{M}_{M_3}}) < 0$ and the secret bit is actually 1.

Thus,

$$\begin{aligned}\Pr(\text{Success}) &= \Pr[nb_i = d_i] \\ &= \Pr[nb_i = 0 \wedge d_i = 0] + \Pr[nb_i = 1 \wedge d_i = 1] \\ &= \Pr[nb_i = 0 \mid d_i = 0] \cdot \Pr[d_i = 0] \\ &+ \Pr[nb_i = 1 \mid d_i = 1] \cdot \Pr[d_i = 1]\end{aligned} \tag{3}$$

If $d_i = 0$, we replace $B_{m,i+1} = t^0_{m,i+1}$ in Equation 1 as,

$$\begin{aligned}\Pr[nb_i = 0 \mid d_i = 0] &= \Pr[(r_{m_4,i+1} \neq t^0_{m_4,i+1}) \wedge (r_{m_3,i+1} = t^0_{m_3,i+1}) \\ &\wedge (r_{m_2,i+1} = t^0_{m_2,i+1}) \wedge (r_{m_1,i+1} \neq t^0_{m_1,i+1})] \\ &= \Pr[(r_{m_4,i+1} \neq t^0_{m_4,i+1}) \wedge (r_{m_3,i+1} = t^0_{m_3,i+1}) \\ &\wedge (r_{m_2,i+1} \neq t^1_{m_2,i+1}) \wedge (r_{m_1,i+1} = t^1_{m_1,i+1})] \\ &(\text{since } t^0_{m_2,i+1} \neq t^1_{m_2,i+1} \text{ and } t^1_{m_1,i+1} \neq t^0_{m_1,i+1})\end{aligned} \tag{4}$$

Substituting the events from Offline Phase,

$$\begin{aligned}\Pr[nb_i = 0 \mid d_i = 0] &= \Pr[(r_{m_4,i+1} = p_{m_4,i+1}) \wedge (r_{m_3,i+1} = p_{m_3,i+1}) \\ &\wedge (r_{m_2,i+1} = p_{m_2,i+1}) \wedge (r_{m_1,i+1} = p_{m_1,i+1})] \\ &= \Pr[(r_{m,i+1} = p_{m,i+1})]\end{aligned} \tag{5}$$

Similar calculations reveal,

$$\Pr[nb_i = 1 \mid d_i = 1] = \Pr[(r_{m,i+1} = p_{m,i+1})] \tag{6}$$

Thus, combining equations we get,

$$\begin{aligned} \Pr(\text{Success}) &= \Pr[r_{m,i+1} = p_{m,i+1}] \cdot [\Pr(d_i = 0) + \Pr(d_i = 1)] \\ &= \Pr[r_{m,i+1} = p_{m,i+1}] \end{aligned} \tag{7}$$

Thus we conclude from this that the probability of success is equal to the probability that the theoretical predictor closely models the real predictor.

6 Experimental Validation for the Online Phase of the Attack

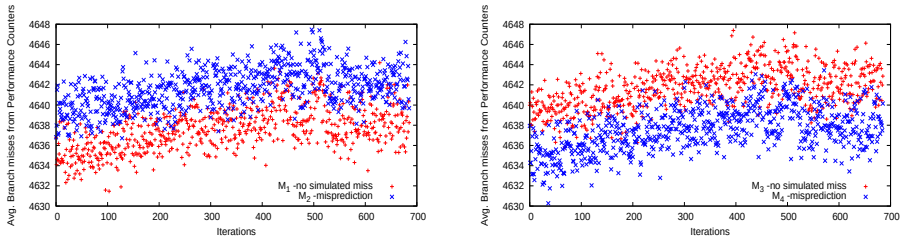
In this section we present the validation of previous discussion through experiments. The experiments are performed on RSA algorithm, the exponents being 1024 bits. The experiments are performed on various Intel processors like Intel Core-2 Duo E7400, Intel Core i3 M350 and Intel Core i5-3470. We illustrate our results by varying following parameters:

- Branch misses from performance counters are captured from the statistic reported by the Perf tool for executables running Square and Multiply algorithm and Montgomery Ladder algorithm using Montgomery multiplication subroutine for performing squaring and multiplications.
- The exponentiations are computed for random inputs of 64 bits that are randomly chosen.
- The performance counter measurements are observed over say L number of inputs. In between every iteration, we perform dummy exponentiation with randomly generated key-bits to flush the effect of the previous iterations from the predictor.
- The entire process is repeated for I number of iterations.

The offline phase of the attack separates a big pool of random inputs M into sets M_1, M_2, M_3 and M_4 based on mispredictions being simulated and results are furnished using 2-bit prediction as well as two-level adaptive predictor.

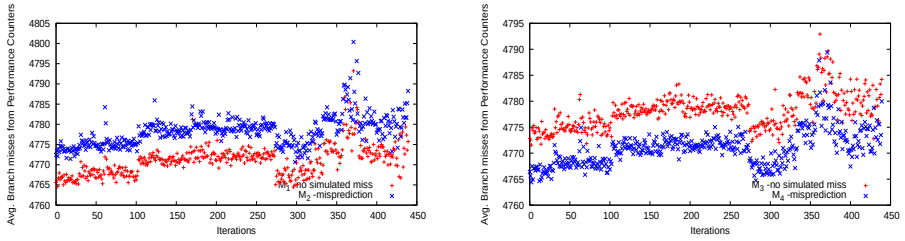
6.1 Experiments on Square and Multiply and Montgomery Ladder Algorithm

Initially the attack is performed on the square and multiply exponentiation implementation targeting the conditional reduction of the $(i+1)^{th}$ squaring step. Figure 4 shows the correct and incorrect separations for all 4 sets (separated by simulations over two-level adaptive predictor) for the randomly chosen 548^{th} bit location of the target key-stream. Figure 4(a) plots average branch misses observed from performance counters for each elements in set M_1 and M_2 (each set having L = 1000 elements) and the experiment is repeated over I = 1000 iterations in order to check the consistency of the output. It is evident from the figure that in most of the iterations the average branch miss for set M_2 is more than the branch misses for set M_1(as expected). On the contrary, Figure 4(b) plots average branch misses observed from performance counters for each elements in set M_3 and M_4. But we observe an incorrect separation as in most of this case, ciphertexts in set M_4 is having lesser branch misses than in set M_3 which is incorrect since theoretically it should be the reverse. Thus from this two



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

Fig. 4. 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



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

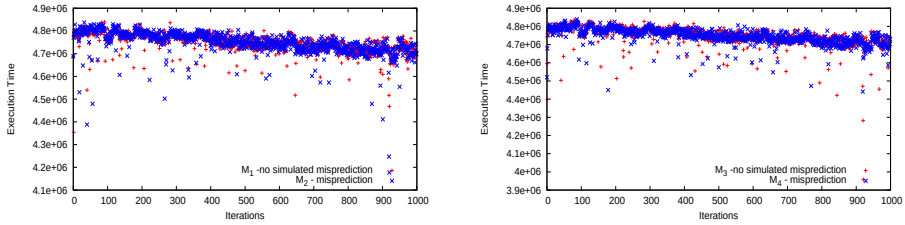
Fig. 5. Branch misses from HPCs on Montgomery Ladder correctly identifies secret bit $d_i = 1$, ciphertext set partitioned by simulated misses of two-level adaptive predictor

figures, the correct exponent can be easily identified showing correct difference in branch misses.

The offline phase for Montgomery Ladder implementation slightly differs from the square and multiply algorithm as appears in Section 4.2. The separation of inputs are performed based on two separate subsimulated traces, and the misprediction is simulated selecting one of them depending on the assumption of the secret bit. The online phase of the attack is carried out similar to the previous experiment having L = 1000 and I = 1000. Figure 5(a), 5(b) shows the correct and incorrect assumptions of the target location for all the 4 ciphertext sets (separated by simulations over two-level adaptive predictor), which illustrates that it can identify the target secret bit correctly. In the following subsection, timing is used as side-channel instead of branch miss in the same experimental scenario but unlike branch miss, timing information fails to reveal the secret bit.

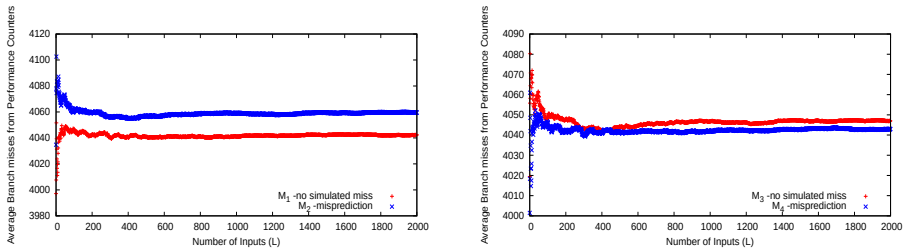
6.2 Comparing timing as side-channel to branch misses from HPC

Timing side-channel as compared to branch misses will require significantly larger number of random inputs so that the adversary can identify next bit correctly. To establish our claim, similar experiments as previous has been experimented with parameters $L = 1000$ and $I = 1000$ and the execution time of the exponentiations over the entire secret key is monitored. Figure 6(a), (b) illustrates that there is no clear demarcation so as to identify the secret bit. The timing side-channel has to be observed on significantly huge number of inputs to observe the accuracy that the adversary is able to observe using branch misses from HPCs. Thus we conclude that branch misses from HPCs can be viewed as



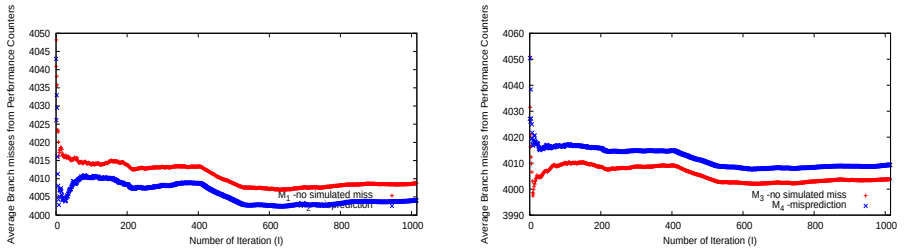
(a) **Correct Assumption** $d_i = 1$ (b) **Incorrect Assumption** $d_i = 0$

Fig. 6. No identification of secret bit is possible using timing as side-channel with $L = 1000$ and $I = 1000$



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

Fig. 7. Variation in the separation of branch misses for correct secret bit = 1 showing positive difference for M_1 and M_2 with the increase in number of ciphertexts(L), I = 100



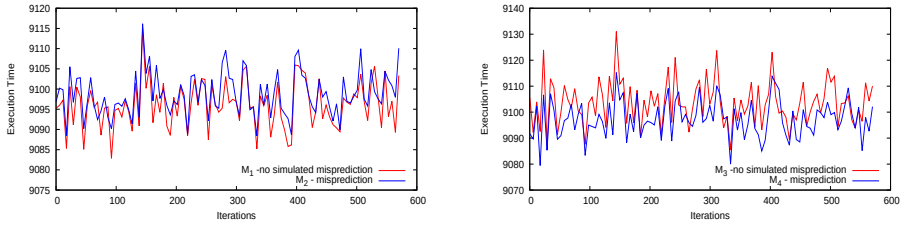
(a) **Incorrect Assumption** $d_i = 1$ (b) **Correct Assumption** $d_i = 0$

Fig. 8. Variation in the separation of branch misses for correct secret bit = 0 showing positive difference for M_3, M_4 with the increase in number of iteration(I), L = 1000

stronger side-channel while exploiting the vulnerabilities of public-key ciphers.

6.3 Variation of parameters such as Number of Inputs (L) and Iteration (I)

Figure 7, 8 shows the variation of the differences in branch misses for the 4 ciphertext sets respectively. In these experiments the ciphertext sets are separated by simulation from the 2-bit dynamic predictor. Thus, from the experimental results as illustrated in Figure 7, 8, we can conclude that the identification of secret bit requires reasonably smaller number of inputs(L) (compared to timing side-channel) and the results are consistent across several iterations(I).



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

Fig. 9. Branch misses from HPCs on RSA-OAEP implementation, correctly identifies secret bit $d_i = 1$, ciphertext set partitioned by simulated misses of bimodal predictor

6.4 Revealing Secret Exponent in RSA-OAEP Randomized Padding Procedure

In this section, we adapt the attack model described in the Section 4 to reveal the secret exponent in the RSA-OAEP padding procedure. A brief description of the padding scheme is presented in Section 2.2 and we present its vulnerabilities with respect to the present attack scenario. In the decryption process of this randomized padding scheme ciphertexts are decrypted using modular exponentiation algorithms over the secret exponent and this decrypted string is decoded while the algorithm outputs the original communicated message if and only if no error has been detected while decoding.

In this paper we target the decryption phase of the RSA-OAEP algorithm. The correctness check on the decrypted input is done after the exponentiation over the secret exponent has been performed. The entire decryption and decoding is operated over a set of randomly generated ciphertexts which may not output valid messages(as they might fail to satisfy all criteria to output valid message). But in this process of the exponentiation operation, the unknown secret exponent gets leaked from the information of branch misses. The offline phase as in Algorithm 4 can be constructed for each ciphertext from the randomly generated set and the online measurements of branch misses over the separate sets eventually reveals the correct guess corresponding to the secret exponent.

We performed the experiments with a Montgomery Ladder implementation of RSA decryption followed by the $RSA\_padding\_check\_PKCS1\_OAEP()$ function from the OpenSSL 1.0.0 library which performs the RSA-OAEP decoding. The experimental results for the RSA-OAEP decryption procedure is illustrated in Figure 6.4(a) and 6.4(b) which clearly shows that for the actual secret bit there is a correct separation while incorrect separation can be observed for the wrong guess. Thus it be stated that, even though Randomized message padding encryptions make ciphers semantically secure, it cannot guarantee security against this attack as the side-channel leakage through branch miss event can be intelligently exploited to reveal the individual key bits one after another, while the exponentiation operation is being performed on the secret exponent for each randomly generated ciphertexts.

7 Discussions

Branching and conditional statements has been first targeted by side-channel cryptanalysts exploiting timing as side-channel. There has been several countermeasures like fuzzying timestamp counters, constant time implementations which have been proposed in literature to thwart the attacks from timing variations. But in most of these countermeasures, threat exists through HPCs as side-channel since the sequence of conditional statements that are being executed remains dependent on the key bits. In the present work though we have illustrated the attacks on RSA-like asymmetric key ciphers but this work can be extended to standard Double and Add Algorithm which is used to implement Elliptic Curve Scalar Multiplication. This forms the basis of the future scope of the study. We propose some of the feasible algorithmic countermeasures which are capable to thwart the present attack:

- Our attack targets the conditional reduction statement of Montgomery Multiplication(MM) and identifies secret key bits on observing branch miss distribution over separate ciphertext sets. If input to MM algorithm is masked such that 2 random numbers are generated at runtime and inputs are modified as $(a_r = a + r_1)$ and $(b_r = b + r_2)$, the branch predictor observes conditional branches which depend on r_1, r_2. However the final product is $a * b$ as effect of r_1, r_2 can be nullified by adding correction terms. This masking strategy will prevent the adversary from simulating branch miss, since r_1, r_2 are randomly generated at run time.
- There are other implementations of RSA, like CRT-RSA, which can be more resistant against the proposed attacks, since the adversary cannot perform the necessary subsimulations without knowing the prime factors of the RSA modulus.

However in context to such implementations, the performance counters can still pose a threatening side-channel, if stronger attack models are considered. For example, if the adversary is capable of introducing a transient bit fault in the secret exponent, and observes the differences in the values of the performance counters, leakages due to the branch predictor still occurs. The idea of the fault attack is such that the difference of branch misses for exponentiation between the original and the faulty key(fault being introduced at the i^{th} bit) can uniquely reveal the subsequent key bits from i^{th} location. The simulated branch miss values from a 2-bit predictor algorithm yields a difference in branch miss(Δ_i) in the range $[-3:3]$ and each of this Δ_i reveals the subsequent key bit pattern given the prior knowledge of the predictor state(St_{i-1}) after $i-1$ bits. (The attack idea is described in brief in Appendix B.)

All these experiments, show that HPCs form a threatening side-channel for the existing implementations of RSA-like public key ciphers and any such implementation which has branching statements conditioned on secret key bits are vulnerable to attacks exploiting branch misprediction information from HPCs. This side-channel should also be considered along with other well-known side-channels like timing, power, and faults. The information provided by the Performance Counters should be possibly computed to provide the user means to access the performance, without providing a mechanism to extract secret information.

8 Conclusion

This paper shows that HPCs, which are used as performance monitors (watchmen) in modern computer systems can be utilized to retrieve the secret keys by reasonably modelled adversaries. The attack that we illustrate exploit the characteristics of branch predictor and show formally that the leakage of the key increases with the ability of the attacker to model the predictor more accurately. The experimental results clearly present the correct identification of the secret bits of 1024 bit RSA running on real life Intel platforms. We follow by a claim that branch misses from HPCs are indeed more significant side-channel compared to timing. For future work these experiments should be widened to model secure predictors which will inherently prevent information leakage.

References

1. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, London, UK, 1996. Springer-Verlag.
2. Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.
3. Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *FDTC*, pages 59–67. IEEE Computer Society, 2008.
4. Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2002.
5. Peter L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.
6. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.
7. James Manger. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs 1 v2.0. In *CRYPTO*, pages 230–238, 2001.
8. RSA Security Inc. RSA Laboratories. Rsaes-oaep encryption scheme. 2000.
9. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.
10. Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *MICRO*, pages 51–61, 1991.
11. Agner Fog. The Microarchitecture of Intel and AMD CPU's, An Optimization Guide for Assembly Programmers and Compiler Makers, 2009.

A Some well-known branch predictor architecture in literature [10]

A.1 Dynamic 2-bit Predictor

The state machine of the dynamic predictor is shown in Fig 10. The state machine has four states and either predict the branches to be taken(1) or not taken(0) depending on the history of taken branches. The input to this state machine is the branch sequence, and the output is the branch prediction.

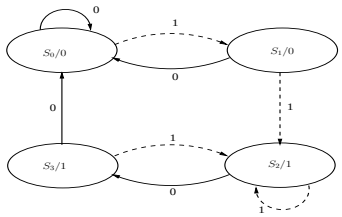


Fig. 10. Dynamic 2-bit Predictor State Machine [9]

The 2-bit dynamic branch predictor as in Figure 10 has four states S_0, S_1, S_2, S_3, each having a predicted output. On an input, the state transitions are denoted with the arrows, and the labels denote the input bit. The predicted output corresponding to each state is noted in the diagram as state S_0, S_1 predicts 0 and state S_2, S_3 predicts 1.

A.2 Two-level Adaptive Predictor

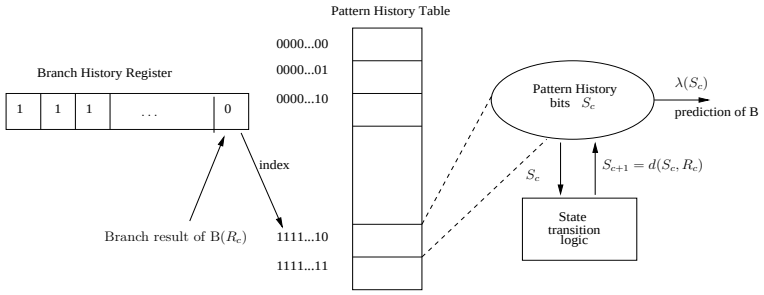


Fig. 11. Two Level Adaptive Branch Prediction [10]

This predictor remembers the last k occurrences of a branch instruction and uses a s-bit prediction function (such as an s-bit predictor) for each of the 2^k history of patterns. *Branch history register* is a shift register of size k, which stores the history of the last k branches – either taken or not-taken. The branch history register indexes into a second level called *pattern history table*, which can hold 2^k entries, each s bits wide. Figure 11 illustrates the two level predictor. When a conditional branch say B is getting predicted,

- content of the k bit history register is used as address to the pattern history table. S_c denotes the contents of the pattern history table.
- S_c is fed to the s-bit prediction decision function (such as the 2-bit dynamic predictor), which outputs the predicted value $\lambda(S_c)$.

B Retrieving Secret from Difference of Branch Misses due to Fault

We describe the fault model for key space of key length n bits on the basis of the following assumptions:

1. The length of the key n bit is known to the adversary.
2. The adversary starts from the Most Significant Bit(always 1), recovering the subsequent bits one bit at a time.
3. A fault is introduced at the i^{th} position of the key sequence which eventually flips the i^{th} bit of the key.
4. The difference in branch misses(Δ_i) for the secret key(K) and the faulty key(F_i) is simulated by using the 2-bit predictor algorithm.

The attack target the event branch miss for the conditional multiplication operation which is being conditioned on the secret exponent bits and the simulated branch misses for exponentiation are not affected by varying the input plaintext or ciphertexts. The simulated branch miss values from a 2-bit predictor algorithm yields a difference in branch miss(Δ_i) in the range $[-3:3]$ and each of this Δ_i reveals the subsequent key bit pattern given the prior knowledge of the predictor state(St_{i-1}) after $i-1$ bits. The tabular representation as in Table 1 demonstrates this scenario and each entry in the table can be combinatorially justified. In the following discussion we try to illustrate one such situation with an example which justifies a single table entry(marked in gray) in Table 1.

Let us assume a small secret key as 1110111011 and a fault is introduced at 5^{th} position from the MSB, thus the faulty key is 1110**0**11011. Note that the branch sequence is the complement of the key sequence for this target implementation. The 5^{th} position of faulty key (from the left) differs from the secret key and let us assume that we already know the first 4 bits as 1110. The branch sequence for the secret key is 0001000100 while for the faulty key is 0001100100. On simulating branch statements (b_0, b_1, b_2, b_3) on a 2-bit predictor for the initial 4 bits, the traces of branch statements are as $\{not-taken, not-taken, not-taken, taken\}$ and the predictor state is at S_1, and thus predicts the next branch as $not-taken$ $(b_4 = 0)$.

Table 1. Branch Decision Sequence Pattern of branches $b_i, b_{i+1}, \cdots, b_{n-1}$ corresponding to bits $d_i, d_{i+1}, \cdots, d_{n-1}$ for One bit Fault

St_{i-1}	Δ_i(Diff. of b.m. of secret & faulty key)						
	-3	-2	-1	**0**	**1**	**2**	**3**
S_0	110	$1(10)^*$	10	_1	00	$0(10)^*$	010
S_1	10	1	-	-	-	0	00
S_2	001	$0(01)^*$	01	_0	11	$1(01)^*$	101
S_3	01	0	-	-	-	1	11

- As the immediate next bit i.e, the 5^{th} bit is 1, for the secret key there is no change in branch miss, but for the faulty key there is a bit flip at the 5^{th} position. So the faulty key suffers from a branch miss increasing the difference by 1.
- At this stage, the predictor states for the correct and faulty key differs. The predicted branch for secret key is $not - taken(0)$(as predictor is in state S_0) but for the faulty key it is $taken(1)$(while in this case, it is in S_1) as it suffered from two mispredictions from two consecutive $taken$ branches(0 bits).
- So if the next two bits are again one then for secret key there is no increase in branch miss.
- But for faulty key, there are two mispredictions due to the mismatch of the predicted value as $taken(1)$ and the bit values at $6, 7^{th}$ position as 1 (where actually the branches are $not - taken$).
- Thus the difference of branch misses between the secret and the faulty key is 3 and the branch sequence (b_i, b_{i+1}) gets revealed as $(0, 0)$ ie, $\{not - taken, not - taken\}$which eventually reveals the key bits $(d_i, d_{i+1}) = (1, 1)$.
- This is the maximum difference because, after getting two consecutive ones at the $6, 7^{th}$ position the predictors branch output for the secret and the faulty key becomes same. Thus further increase in branch misses gets cancelled as they occur in both secret and its faulty counterpart and do not add up to the difference.

Similarly other entries of Table 1 can be justified with respective rationale about the subsequent secret key bits and eventually revealing them. Though all entries in the above said table cannot be determined uniquely with a single-bit fault(denoted by "-" in Table 1), similar analysis can be done on a 2-bit fault model to uniquely determine the remaining entries. This unique classification can be done with difference of branch misses($\Delta_{i,i+1}$) observed from a single fault on $(i+1)^{th}$ bit and 2-bit fault on $i, (i+1)^{th}$ locations.

Once the formal modelling is done, the next step is to validate this approximation with actual branch miss information from HPCs. The real attack on actual systems can be featured with template building and template matching

Table 2. Key Sequence Pattern of b_i, b_{i+1} for Two bit Fault

St_{i-1}^K \ $\Delta_{i,i+1}$	-1	**1**
S_0	11	01
S_2	00	10

phases for each difference(Δ_i) class from actual branch miss information through HPCs. Here, the aim is to learn a correlation between the theoretical difference of branch misses from the 2-bit branch predictor and the differences obtained from actual hardware performance counters in real processors.

A set of random keys are generated. These keys are classified on the basis of the difference of branch mispredictions (Δ_i) from 2-bit predictor algorithm (between original key and its faulty counterpart) into respective classes of difference in the range $[-3, 3]$. Thus multiple keys and their respective faulty key sequences are classified according to observed Δ_i from 2-bit predictor algorithm.

Next by using information of branch mispredictions obtained from HPCs, distributions of branch misses can be constructed for each of the random keys as well as for their faulty counterparts. Thus for all pair of secret and its fault keys belonging to a particular difference class Δ_i, the mean values from this difference distribution for a set of keys (belonging to a specific Δ_i class) forms another distribution of mean differences (over a set of keys having same Δ_i). The mean values for the distributions are tabulated in the Table 3 for respective Δ_i's for a set of random keys of length $n = 1024$ bits, averaged over a set of 1000 random inputs.

Table 3. Tabulating means from hardware performance counters on **Intel Core 2 Duo** for specific $St_{i-1} = S_j$ with respective Δ_i's for key length $n = 21$ bit and $i = 10, j = 0, 1, 2, 3$

St_{i-1}	Δ_i						
	-3	-2	-1	**0**	**1**	**2**	**3**
S_0	0.2201	1.4601	-1.6699	0.0846	1.7751	-1.0907	-0.3539
S_1	-0.085	1.3891	-1.6401	-0.0608	1.3577	-1.4847	0.2337
S_2	0.1584	0.9915	-1.5393	0.3494	1.7987	-0.9951	-0.0741
S_3	0.145	1.9085	-1.5018	0.145	0.9813	-2.4997	0.0741

Thus we observe from Table 3, that the distributions corresponding to $\Delta_i = -1$ and 1 are the distributions which are most separated among the other distributions. Thus from this, we conclude that if $\Delta_i = 1, -1$ for a secret key and the

faulty counterpart from 2-bit predictor algorithm, then adversary will succeed to determine the subsequent bits by observing the distribution of differences in branch misses.