Predicting Secret Keys via Branch Prediction

Onur Acıiçmez[1], Jean-Pierre Seifert[2,3], and Çetin Kaya Koç[1,4]

[1] Oregon State University
School of Electrical Engineering and Computer Science
Corvallis, OR 97331, USA
[2] Applied Security Research Group
The Center for Computational Mathematics and Scientific Computation
Faculty of Science and Science Education
University of Haifa
Haifa 31905, Israel
[3] Institute for Computer Science
University of Innsbruck
6020 Innsbruck, Austria
[4] Information Security Research Center
Istanbul Commerce University
Eminönü, Istanbul 34112, Turkey
aciicmez@eecs.oregonstate.edu, jeanpierreseifert@yahoo.com, koc@cryptocode.net

Abstract. This paper presents a new software side-channel attack — enabled by the branch prediction capability common to all modern high-performance CPUs. The penalty payed (extra clock cycles) for a mispredicted branch can be used for cryptanalysis of cryptographic primitives that employ a data-dependent program flow. Analogous to the recently described cache-based side-channel attacks our attacks also allow an unprivileged process to attack other processes running in parallel on the same processor, despite sophisticated partitioning methods such as memory protection, sandboxing or even virtualization. We will discuss in detail several such attacks for the example of RSA, and experimentally show their applicability to real systems, such as OpenSSL and Linux. More specifically, we will present four different types of attacks, which are all derived from the basic idea underlying our novel side-channel attack. Moreover, we also demonstrate the strength of the branch prediction side-channel attack by rendering the obvious countermeasure in this context *(Montgomery Multiplication with dummy-reduction)* as useless. Although the deeper consequences of the latter result make the task of writing an efficient and secure modular expeonentiation (or scalar multiplication on an elliptic curve) a challenging task, we will eventually suggest some countermeasures to mitigate branch prediction side-channel attacks.

Keywords: Branch Prediction, Modular Exponentiation, Montgomery Multiplication, RSA, Side Channel Analysis, Simultaneous Multithreading, Trusted Computing.

1 Introduction

The contradictory requirement of increased clock-speed with decreased power-consumption for todays computer architectures makes branch predictors an inevitable central CPU ingredient, which significantly determines the so called *Performance per Watt* measure of a high-end CPU, cf. [GRAB]. Thus, it is not surprising that there has been a vibrant and very practical research on more and more sphisticated branch prediction mechanisms, cf. [PH,Sha,She].

Unfortunately, the present paper identifies branch prediction even in the presence of recent security promises for commodity platforms from the Trusted Computing area as a novel and unforseen security risk. Indeed, although even the most recently found security risks for x86-based CPU's have been implicitly pointed out in the old but thorough x86-architecture security analysis, cf. [SPL], we have not been able to find any hint in the literature spotting branch prediction as an obvious side channel

attack victim. Let us elaborate a little bit on this connection between side-channel attacks and modern computer-architecture ingredients.

So far, typical targets of side-channel attacks have been mainly Smart Cards, cf. [CNK,Koc]. This is due to the ease of applying such attacks to smart cards. The measurements of side-channel information on smart cards are almost "noiseless", which makes such attacks very practical. On the other side, there are many factors that affect such measurements on real commodity computer systems based upon the most successful one, the Intel x86-architecture, cf. [Sha]. This factors create noise, and therefore it is much more diffucult to develop and perform successful attacks on such "real" computers within our daily life. Thus, until very recently the vulnerability of systems even running on servers was not "really" considered to be harmful by such side-channel attacks. This was changed with the work of Brumley and Boneh, cf. [BB], who demonstrated a remote timing attack over a real local network. They simply adapted the attack principle as introduced in [Sch] to show that the RSA implementation of OpenSSL [open] — the most widely used open source crypto library — was not immune to such attacks.

Even more recently, we have seen an increased research effort on the security analysis of the daily life PC platforms from the side-channel point of view. Here, it has been especially shown that the cache architecture of modern CPU's creates a significant security risk, cf. [Ber,OST05,OST06,Per], which comes in different forms. Although the cache itself has been known for a long time being a cruical security risk of modern CPU's, cf. [SPL,Hu] the above papers were the first proving such vulnerabilities practically and raised large public interest in such vulnerabilities.

Especially in the light of the ongoing Trusted Computing efforts, cf. [TCG], which promise to turn the commodity PC platform into a trustworthy platform, cf. also [CEPW,ELMP$^+$,Gra,Pea,TCG,UNRS$^+$], the formerly described side channel attacks against PC platforms are very interesting and of particular interest. Even more interesting is the fact that all of the above pure software side channel attacks also allow a totally unprivileged process to attack other processes running in parallel on the same processor (or even remote), despite sophisticated partitioning methods such as memory protection, sandboxing or even virtualization. This particularly means that side channel attacks render all of the sophisticated protection mechanisms as for e.g. described in [Gra,UNRS$^+$] as useless. The simple reason for the failure of these trust mechanisms is that the new side-channel attacks simply exploit deeper processor ingredients — i.e., below the trust architecture boundary, cf. [PL,Gra].

Having said all this, it is natural to identify other modern computer architectural ingredients which have not yet been discovered as a security risk and which are operating below the current trust architecture boundaries. That is the focus of the present paper — a processor's Branch Prediction Unit (BPU). Namely, we will analyze BPUs and highlight the security vulnerabilities associated with their opaque operations deep inside a processor. In other words, we will present so called branch prediction attacks on simple RSA implementations as a case study to describe the basics of the novel attacks an adversary can use to compromise the security of a platform. Our attacks can also be adapted to other RSA implementations and/or other public-key systems like ECC. We will try to refer to specific vulnerable implementations throughout this text.

The paper is organized as follows. We will first give some background information including the structure and functionality of a general BPU and the details of the used RSA-implementations in the next section. Then the following section presents four different attack principles to exploit a BPU in order to break some standard RSA implementations. To do so, we gradually develop from an obvious attack principle more sophisticated attack principles having the potential to break even the implementations that are believed to be immune to side channel attacks. This section is then complemented by presenting the results of some practical implementations of our various attack scenarios. To fully articulate the strength of our attack principles we have only chosen to implement such attack scenarios which are easy to achieve/expect in practice but having greatest practical significance. Hereafter, we draw some conclusions from the the paper and point the reader to further interesting research areas.

2 Background, Definitions and Preliminaries

Although it is benefical — in order to completely understand our attacks as described later — to know many details about modern computer architecture and branch prediction schemes, it would be unrealistic to explain all these subtle details here. Thus, we refer the reader to [PH,Sha,She] for a thorough treatment of this topics. Nevertheless, we will now explain the basic concepts common to any branch prediction unit, although the exact details differ from processor to processor and are not completely explained in freely available processor manuals.

2.1 Branch Prediction Unit

Superscalar processors have to execute instructions speculatively to overcome control hazards, cf. [She]. The negative effect of control hazards on the effective machine performance increases as the depth of pipelines increases. This fact makes the efficiency of speculation one of the key issues in modern superscalar processor design. The solution to improve the efficiency of is to speculate on the most likely execution path. The success of this approach depends on the accuracy of branch prediction. Better branch prediction techniques improve the overall performance a processor can achieve, cf. [She].

A *branch instruction* is a point in the instruction stream of a program where the next instruction is not necessarily the next sequential one. There are two types of branch instructions: unconditional branches (e.g. jump instructions, goto statements, etc.) and conditional branches (e.g. if-then-else clauses, for and while loops, etc.). For conditional branches, the decision to take the branch or not to take depends on some condition that must be evaluated in order to make the correct decision. During this evaluation period, the processor speculatively executes instructions from one of the possible execution paths instead of stalling and awaiting for the decision to come through. Thus, it is very beneficial if the branch prediction algorithm tries to predict the most likely execution path in a branch. If the prediction is true, the execution continues without any delays. If it is wrong, which is called a *misprediction*, the instructions on the pipeline that were speculatively issued have to be dumped and the execution starts over from the mispredicted path. Therefore, the execution time suffers from a misprediction. The misprediction penalty obviuosly increases in terms of clock cycles as the depth of pipeline extends. To execute the instructions speculatively after a branch, the CPU needs the following information:

- *The outcome of the branch.* The CPU has to know the outcome of a branch, i.e., taken or not taken, in order to execute the correct instruction sequence. However, this information is not available immediately when a branch is issued. The CPU needs to execute the branch to obtain the necessary information, which is computed in later stages of the pipeline. Instead of awaiting the *actual* outcome of the branch, the CPU tries to predict the instruction sequence to be executed next. This prediction is based on the history of the same branch as well as the history of other branches executed just before the current branch, cf. [She].
- *The target address of the branch.* The CPU tries to determine if a branch needs to be taken or not taken. If the prediction turns out to be taken, the instructions in the target address have to be fetched and issued. This action of fetching the instructions from the target address requires the knowledge of this address. Similar to the outcome of the branch, the target address may not immediately available too. Therefore, the CPU tries to keep records of the target addresses of previously executed branches in a buffer, the so called *Branch Target Buffer (BTB).*

Overall common to all *Branch Prediction Units (BPU)* is the following Figure 1.

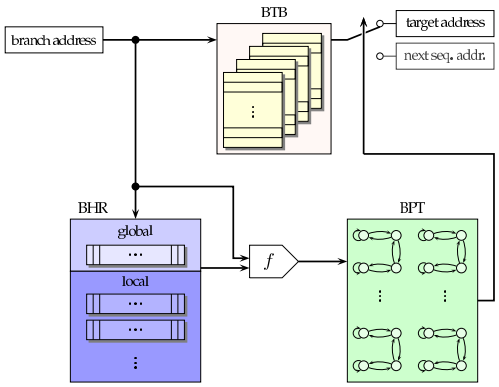

Fig. 1. Branch Prediction Unit Architecture

As shown, the BPU consists of mainly two "logical" parts, the BTB and the predictor. As said already above, the BTB is the buffer where the CPU stores the target addresses of the previous branches. Since this buffer is limited in size, the CPU can store only a number of such target addresses, and previously stored addresses may be evicted from the buffer if a new address needs to be stored instead.

The predictor is that part of the BPU that makes the prediction on the outcome of the branch under question. There are different parts of a predictor, i.e., Branch History Registers (BHR) like the global history register or local history registers, and branch prediction tables, cf. [She].

2.2 Details of popular RSA Implementations

RSA is the most widely used public key cryptosystem which was developed by Rivest, Shamir and Adleman, cf. [MvOV]. The main computation in RSA decryption/signing is the modular exponentiation $P = M^d (\mathrm{mod} N)$, where M is the message or ciphertext, d is the private key that is a secret, and N is the public modulus. Here, N is a product of two large primes p and q. If an adversary obtains the secret value d, he can read all encrypted messages and impersonate the owner of the key. Therefore, the usual main purpose of using timing attacks is to reveal this secret value. If the attacker can factorize N, i.e., he can obtain either p or q, the value of d can be easily calculated. Hence, the attacker tries to find p, q, or d. Since the size of the key is very large, e.g., 1024 bits, the exponentiation is very expensive in terms of the execution time. Therefore, actual implementations of RSA employ efficient algorithms to calculate the result of this operation. In the next subsections we explain the most widely used algorithms, which can be exploited by various timing attacks.

Binary Square-and-Multiply Exponentiation Algorithm. The binary version of Square-and-Multiply Algorithm (SM) is the simplest way to perform an exponentiation. We want to compute $M^d (mod\ N)$, where d is an n-bit number, i.e., $d = (d_0, d_1, ..., d_{n-1})_2$. Figure 2 shows the steps of SM, which processes the bits of d from left to right.

The reader should note that all of the multiplications and squarings are shown as modular operations, although basic SM algorithm computes regular exponentiations. This is because RSA performs modular exponentiation, and our focus here is only on RSA. In an efficient RSA implementation all of the multiplications and squarings are actually performed using a special modular multiplication algorithm, so called Montgomery Multiplication, which we will also recall now.

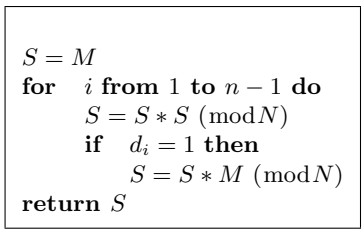

Fig. 2. Binary version of Square-and-Multiply Exponentiation Algorithm

Montgomery Multiplication. Montgomery Multiplication (MM) is the most efficient algorithm to compute a modular multiplication [MvOV]. It simply uses additions and divisions by powers of 2, which can be accomplished by shifting the operand to the right. Since it eliminates time consuming integer divisions, the efficiency of the algorithm is superior to straightforward school-book methods. Montgomery Multiplication is used to calculate $Z = A * B * R^{-1} \pmod{N}$, where A and B are the "N-residues" of a and b with respect to R, R is a constant power of 2, and R^{-1} is the inverse of R in modulus N. I.e., $A = a * R \pmod{N}$, $B = b * R \pmod{N}$, $R^{-1} * R = 1 \pmod{N}$, and $R > N$. Another constraint is that R has to be relatively prime to N. But since N is a product of two large primes in RSA, choosing an R of a power of 2 is sufficient to guarantee that these two numbers are relatively prime. Let N be a k-bit odd number, then 2^k is the most suitable value for R. A conversion to and from N-residue format is required to use this algorithm. Hence, it is more attractive to be used for repeated multiplications on the same residue, just like modular exponentiations. Figure 3 shows the steps of Montgomery Multiplication Algorithm.

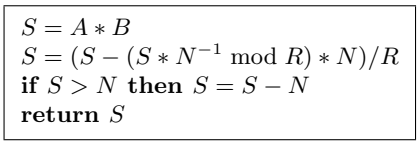

Fig. 3. Montgomery Multiplication Algorithm

The conditional subtraction $S - N$ on the third line is called "extra reduction" and is in general of paticular interest for Side Channel Attacks, cf. [DKLMQ,Sch].

3 Outlines of the various Attack Principles

We will gradually develop 4 different attack principles in this section. Although we describe these attacks on a simple RSA implementation, the underlying ideas can be used to develop similar attacks on different implementations of RSA and/or on other ciphers based upon ECC. In order to do so we will assume that an adversary knows every detail of the BPU architecture as well as the implementation details of the cipher (Kerckhoffs' Principle). This is indeed a valid assumption as the BPU details can be extracted using some simple benchmarks like the ones given in [MMK].

3.1 Attack 1 — Exploiting the Predictor directly (Direct Timing Attack)

In this attack, we rely on the fact that the prediction algorithms are deterministic, i.e., the prediction algorithms are predictable. We present a simple attack below, which demonstrates the basic idea behind this attack. The presented attack is a modified version of Dhem et al.'s attack [DKLMQ].

Assume that the RSA implementation employs Square-and-Multiply exponentiation and Montgomery Multiplication. Assume also that an adversary knows the first i bits of d and is trying to reveal d_i. For any message m, he can simulate the first i steps of the operation and obtain the intermediate result that will be the input of the $(i+1)^{\text{th}}$ squaring. Then, the attacker creates 4 different sets M_1, M_2, M_3, and M_4, where

$M_1 = \{m \mid m \text{ causes a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 1\}$
$M_2 = \{m \mid m \text{ does not cause a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 1\}$
$M_3 = \{m \mid m \text{ causes a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 0\}$
$M_4 = \{m \mid m \text{ does not cause a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 0\}$,

MM means Montgomery Multiplication. If the difference between timing characteristics, e.g., average execution time, of M_1 and M_2 is more significant than that of M_3 and M_4, then he guesses that $d_i = 1$. Otherwise d_i is guessed to be 0. To express the above idea more mathematically, we define:

- An Assumption A_t^i: $d_i = t$, where $t \in \{0, 1\}$.
- A Predicate $\mathbf{P} : (m) \rightarrow \{0, 1\}$ with

$$\mathbf{P}(m) = \begin{cases} 1 & \text{if a misprediction occurs during the computation of } m^2 (\text{mod} N) \\ 0 & \text{otherwise.} \end{cases}$$

- An Oracle $\mathbf{O}_t : (m, i) \rightarrow \{0, 1\}$ under the assumption A_t^i:

$$\mathbf{O}_t(m, i) = \begin{cases} 1 & \mathbf{P}(m_{temp}) = 1 \\ 0 & \mathbf{P}(m_{temp}) = 0, \end{cases}$$

  where $m_{temp} = m^{(d_0, d_1, \ldots d_{i-1}, t)_2} (\text{mod} N)$.
- A Separation $\mathbf{S}_t^i$ under the assumption A_t^i:

$$(S_0, S_1) = (\{m | \mathbf{O}_t(m, i) = 0\}, \{m | \mathbf{O}_t(m, i) = 1\}).$$

For each bit of d, starting from d_1, the adversary performs two partitionings based on the assumptions A_0^i and A_1^i, where d_i is the next unknown bit that he wants to predict. He partitions the entire sample into two different sets. Each assumption and each plaintext, M, in one of these sets yields the same result for $\mathbf{O}_t(M, i)$. We call these partitioning separations $\mathbf{S}_0^i$ and $\mathbf{S}_1^i$. Depending on the actual value of d_i, one of the assumptions A_0^i and A_1^i will be correct. We define the separation under the correct assumption as "Correct Separation" and the other as "Random Separation". I.e., we define the Correct Separation CS_i as

$$(S_0^C, S_1^C) = (\{M | \mathbf{O}_t(M, i) = 0\}, \{M | \mathbf{O}_t(M, i) = 1\}) \text{ where } d_i = t,$$

and the Random Separation RS_i as

$$(S_0^R, S_1^R) = (\{M | \mathbf{O}_t(M, i) = 0\}, \{M | \mathbf{O}_t(M, i) = 1\}) \text{ where } d_i \neq t.$$

The encryption of each plaintext in S_1^C encounters a misprediction delay during the i^{th} squaring, whereas none of the plaintext in S_0^C results in a misprediction during the same computation. Therefore, the adversary will realize a significant timing difference between these two sets and he can predict the value of d_i. On the other hand, the occurrences of the mispredictions will be random-like for the sets S_0^R and S_1^R, which is the reason why we call it a random separation. We can define a correct decision as taking that decision $d_i = t$, where $\mathbf{O}_t(M, i) = \mathbf{P}(M^{(d_0, d_1, \ldots d_i)_2} (\text{mod} N), i)$ for each possible M.

This attack requires the knowledge of the BPU state just before the encryption, since this state, as well as the execution of the cipher, determines the prediction of the target branch. This information is not readily available to an adversary. However, he can perform the analysis phase assuming each possible state one at a time. One expects that only under the assumption of the correct state the above separations should yield a significant difference. Yet, a better approach is it to set the BPU

state manually. If the adversary has access to the machine the cipher is running on, he can execute a process to reset the BPU state or to set it to a desired state. This will be the strategy for our other attacks. This type of attacks can be applied on any platform as long as a deterministic branch prediction algorithm is used on it. To break a cipher using this kind of attack, we need to have a target branch, an outcome which must depend on the secret/private key of the cipher, a known nonconstant value like the plaintext or the ciphertext, and (possibly) some unknown values that can be searched exhaustively in a reasonable amount of time.

Examples of vulnerable systems. RSA with MM and without CRT (Chinese Remainder Theorem) are susceptible to this kind of attack. The conditional branch of the extra reduction step can be used as the target branch. We have already showed the attack on S&M exponentiation. It can be adapted to b-ary and sliding windows exponentiation, cf. [MvOV], too. In these cases, the adversary needs to search each window value exhaustively and construct the partitions for each of these candidate window values. He encounters the correct separation only for the correct candidate and therefore can realize the correct value of the windows. If CRT is employed in the RSA implementation, we cannot apply this attack. The reason is that the outcome of the target branch will also depend on the values of p and q, which are not feasible to be searched exhaustively. Similarly, if the RSA implementation does not have a branch that is taken or not taken depending on a known nonconstant value (e.g. extra reduction step in Montgomery Multiplication, which is input dependent to be performed), we cannot use this approach to find the secret key. For example, the if statement in S&M exponentiation (c.f. Line 4 in Fig. 2) as our target branch is not vulnerable to this attack. This is due to the fact that the mispredictions will occur in exactly the same steps of the exponentiation regardless of the input values, and one set in each of the two separations will always be empty.

3.2 Attack 2 — Forcing the BPU to the Same Prediction (Asynchronous Attack)

In this attack we assume that the cipher runs on a simultaneous multi-threading (SMT) machine, cf. [She], and the adversary can run a dummy process simultaneously with the cipher process. In such a case, he can clear the BTB via the operations of the dummy process and causes a BTB miss during the execution of the target branch. The BPU automatically predicts the branch not to be taken if it misses the target address in the BTB. Therefore, there will be a misprediction whenever the actual outcome of the target branch is 'taken'. We stress that the two parallel threads are isolated and share only the common BPU resource, cf. [She,Sha,OST06,Per]. Borrowed from [OST06] we named this kind of attack an Asynchronous Attack, as the adversary-process needs *no* synchronization with the simultaneous crypto process. Here, an adversary also does not need to know any detail of the prediction algorithm. He can simulate the exponentiations as done in the previous attack and can partition the sample based on the "actual" outcome of the branch. In other words, the following predicate in the oracle (c.f. Section 3.1) can be used:

$$\mathbf{P}(m) = \begin{cases} 1 & \text{if the target branch is taken during the computation of } m^2 (\mathrm{mod} N) \\ 0 & \text{otherwise.} \end{cases}$$

The adversary does not have to clear the whole BTB, but only that BTB set that stores the target address of the branch under consideration, i.e., the target branch. We define three different ways to achieve this:

- *Total Eviction Method:* the adversary clears the whole BTB continuously.
- *Partial Eviction Method:* the adversary clears only a part of the BTB continuously. The BTB set that stores the target address of the target branch has to be in this part.
- *Single Eviction Method:* the adversary continuously clears only the single BTB set that stores the target address of the target branch.

The easiest method to apply, is clearly the first one, because an adversary does not have to know the specific address of the target branch. Recall that the set of the BTB to store the target address of a branch is determined by the actual logical address of that branch. The resolution of clearing the BTB plays a crucial role in the performance of the attack. We have assumed so far that it was possible to clear the entire BTB between two consecutive squaring operations of an exponentiation. However, in practice this is not (always) the case. Clearing the whole BTB may take more time than it takes to perform the operations between two consecutive squarings. Although, this does not nullify the attack, it will mandate (most likely) a larger sample size. Therefore, if an adversary can apply one of the last two eviction methods, he can improve the performance of the attack. There is indeed a way to determine a smaller (than the entire BTB) group of the possible BTB sets that can store the record of the target branch. We will explain this methodology in Section 3.4. But we want to mention that, from the cryptography point of view. we can assume that an adversary knows the actual address of any branch in the implementation due to Kerckhoff's Principle. Under this assumption, the adversary can apply the single eviction method and achieve a very low resolution, which enables him to cause a BTB miss each time the target branch is executed. Recall also, that there is no complicated synchronization between crypto and adversaty process needed.

Examples of vulnerable systems. The same systems that are vulnerable to the first attack (c.f. Section 3.1) are also vulnerable to this kind of attack. The main difference of this attack compared to the first one is the ease of applying it, i.e., unnecessity of knowing = reverse-engineering the subtle BPU details, yielding the correct BPU states for specific time points.

3.3 Attack 3 — Forcing the BPU to the Same Prediction (Synchronous Attack)

In the previous attack, we have specifically excluded the synchronization issue. However, if the adversary finds a way to establish a synchronization with the cipher process, i.e., he can determine for (e.g.) the i^{th} step of the exponentiation and can clear the BTB just before the i^{th} steps, then he can introduce misprediction delays at certain points during the computation. Borrowed again from [OST06] we named this kind of attack a Synchronous Attack, as the adversary-process needs some kind of *synchronization* with the simultaneous crypto process. Assume that the RSA implementation employs S&M exponentiation and the if statement in S&M exponentiation (c.f. Line 4 in Fig. 2) is used as the target branch. As stated above, the previous attacks cannot break this system if only the mentioned conditional branch is examined. However, if the adversary can clear the BTB set of the target branch (c.f. Single Eviction Method in Section 3.2) just before the i^{th} step, he can directly determine the value of d_i in the following way.

The adversary runs RSA for a *known* plaintext and measures the execution time. Then he runs it again for the same input but this time he clears the single BTB set *during* the encryption just before the i^{th} execution of the conditional branch under examination, i.e., the if statement of Line 4 in Fig. 2. This conditional branch is taken or not taken depending only on the value of d_i. If it turns out to be taken, the second encryption will take longer time than the first execution because of the misprediction delay. Therefore, the adversary can easily determine the value of this bit by successively analyzing the execution time.

Examples of vulnerable systems. Any implementation of a cryptosystem is vulnerable to this kind of attack if the execution flow is "key-dependent". The exponents of RSA with S&M exponentiation can be directly obtained even if the CRT is used. If RSA employs sliding window exponentiation, then we can find a significant number of bits (but not all) of the exponents. However, if b-ary method is employed, then only 1 over 2^{wsize} of the exponent bits can be discovered, where $wsize$ is the size of the window. This attack can even break such prominent and efficient implementations that had been considered to be immune to certain kinds of side-channel attacks, cf. [JY,Wal].

3.4 Attack 4 — Trace-driven Attack against the BTB (Asynchronous Attack)

In the previous three attacks, we have considered analyzing the execution time of the cipher. In this attack, we will follow a different approach. Again, assume that an adversary can run a spy process simultaneously with the cipher. This spy process continuously executes unconditional branches and all of these branches map to the same BTB set with the conditional branch under attack. In other words, there is a conditional branch (under attack) in the cipher, which processes the exponent and executes the corresponding sequence of operations. Moreover, assume also that the branches in the spy process and the cipher process can only be stored in the same BTB set. The number of branches in the spy process needs to be equal to the number of ways of associativity of BTB. Recall that it is easy to understand the properties of the BTB using simple benchmarks as explained in [MMK].

The adversary starts the spy process before the cipher, so when the cipher starts encryption/signing, the CPU *cannot* find the target address of the target branch in BTB and the prediction *must* be *not-taken*, cf. [She]. If the branch turns out to be taken, then a misprediction will occur and the target address of the branch needs to be stored in BTB. Then, one of the spy branches has to be evicted from the BTB so that the new target address can be stored in. When the spy-process re-executes his branches, he will encounter a misprediction on the branch that has just been evicted. If the spy-process also measures the execution time of his branches (altogether), then he can detect whenever the cipher modifies the BTB, meaning that the execution time of these spy branches takes a little longer than usual. Thus, the adversary can simply determine the complete execution flow of the cipher process by continuously performing the same operations, i.e., just executing spy branches and measuring the execution time. He will see the prediction/misprediction trace of the target branch, and so he can determine the execution flow. Borrowed again from [OST06] we named this kind of attack an Asynchronous Attack, as the adversary-process needs *no* synchronization at all with the simultaneous crypto process — he is just following his paradigm: continously execute spy branches and measure their execution time.

Examples of vulnerable systems. Any implementation that is vulnerable to the previous attack is also vulnerable to this one. Specifically any implementation of a cryptosystem is vulnerable to this kind of attack if the execution flow is "key-dependent". This attack, on the other hand, is very easy to apply, because the adversary does not have to solve the synchronization problem at all. Considering all these aspects of the current attack, we can confidently say that it is a powerful and practical attack, which puts many of the current public-key implementations in danger.

4 Practical Results

We also performed practical experiments to validate the aforementioned attacks which exploit the branch predictor behavior of modern microprocessors. Obviously, eviction-driven attacks using simultaneous multithreading are more general, and demand nearly no knowledge about the underlying BPU — compared to the other type of branch prediction attacks from above. Thus, we have chosen to carry out our experimental attacks in a popular simultaneous multithreading environment, cf. [Sha]. In this setting an adversary can apply this kind of attacks without any knowledge on the details of the used branch prediction algorithm and BTB structure. Therefore we decided to implement our two asynchronous attacks and show their results as a "proof-of-concept".

4.1 Results for attack 2 = Forcing the BPU to the Same Prediction (Asynchronous Attack)

In this kind of attack we have chosen for reasons of simplicity and practical significance to implement the total eviction method. We used a dummy process that continuously evicts BTB entries by executing conditional branches. This process was simultaneously running with RSA on an SMT platform. It

executed a large number of conditional branches and evicted each single BTB entry one at a time. This method requires almost no information on the BTB structure. We performed this attack on a very simple RSA implementation that employed square-and-multiply exponentiation and Montgomery multiplication *with* dummy reduction. We used the RSA implementation in OpenSSL version 0.9.7e as a template and made some modifications to convert this implementation into the simple one as stated above. To be more precise, we changed the window size from 5 to 1, turned blinding off, removed the CRT mode, and added the dummy reduction step. The source code was compiled using the gcc compiler with default options. The experiments were run under the configuration shown in Table 1. We used random plaintexts generated by the `rand()` and `srand()` functions available in the standard C library. The current time was fed into `srand()` function as the pseudorandom number generation seed. We measured the execution time in terms of clock cycles using the cycle counter instruction `RDTSC`, which is available in user-level.

Operating System:	RedHat workstation 3
Compiler:	gcc version 3.2.3
Cryptographic Library:	OpenSSL 0.9.7e

Table 1. The configuration used in the experiments

We generated 10 million random single-block messages and measured their encryption times under a fixed 512-bit randomly generated key. In our analysis phase, we eliminated the outliers and used only 9 million of these measurements. We then processed each of these plaintext and divided them into the sets as explained in Section 3.1 and Section 3.2 based on the assumption of the next unknown bit and the assumed outcome of the target branch. Hereafter we calculated the difference of the average execution time of the corresponding sets for each bit of the key except the first two bits. The mean and the standard deviation of these differences for correct and random separations are given in the following Figure 4. This figure shows also on the right side, the raw timing differences after averaging the 9 million measurements into one single timing difference, where a single dot corresponds to the timing difference of a specific exponent bit, i.e., the x-axis corresponds to the exponent bits from 0 to 511.

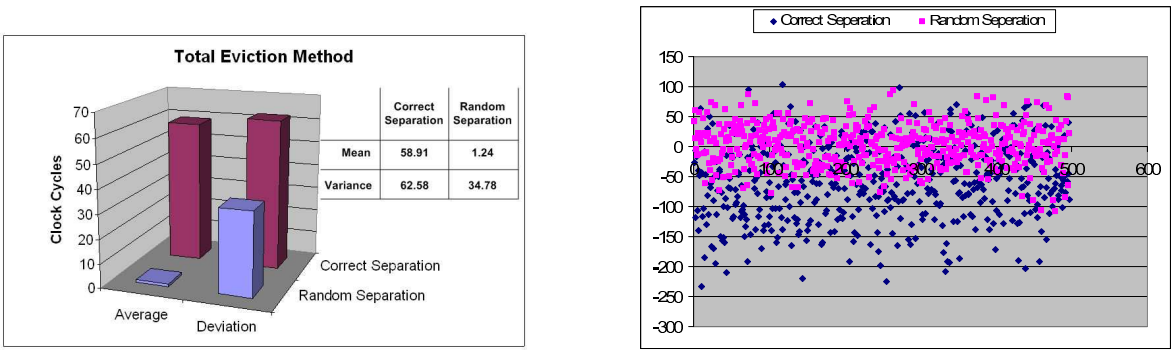

Fig. 4. Practical results when using the total eviction method in attack principle 2.

Using the values in Figure 4, we can calculate the probability of successful prediction of any single key bit. We interpret the measured average execution time differences for correct and random separation as realizations of normally (Gaussian) distributed random variables, denoted by Y and X respectively. We may assume $Y \sim N(\mu_Y, \sigma_Y^2)$ and $X \sim N(\mu_X, \sigma_X^2)$ for each bit of any possible key, where $\mu_Y = 58.91$, $\mu_X = 1.24$, $\sigma_Y = 62.58$, and $\sigma_X = 34.78$, cf. Figure 4. We then introduce the normally distributed random variable Z as the difference between realizations of X and Y, i.e.,

$Z = Y - X$ and $Z \sim N(\mu_Z, \sigma_Z^2)$. The mean and deviation of Z can be calculated from those of X and Y as

$$\mu_Z = \mu_Y - \mu_X = 58.91 - 1.24 = 57.67$$
$$\sigma_Z = \sqrt{\sigma_Y^2 + \sigma_X^2} = \sqrt{(62.58)^2 + (34.78)^2} = 71.60$$

As our decision strategy is it to pick that assumption of the bit value that yields the highest execution time difference between the sets we constructed under that assumption, our decision will be correct whenever $Z > 0$. The probability for this realization, $\Pr[Z > 0]$, can be determined by using the z-distribution table, i.e.,

$$\Pr[Z > 0] = \Phi((0 - \mu_Z)/(\sigma_Z) = \Phi(-0.805) = 0.79,$$

which shows that our decisions will be correct with probability 0.79 if we use $N = 10$ million samples. Although we could increase this accuracy by increasing the sample size, this it is not necessary. If we have a wrong decision for a bit, both of the separations will be random-like afterwards and we will only encounter relatively insignificant differences between the separations. Therefore, it is possible to detect an error and recover from a wrong decision without necessarily increasing the sample size. More importantly, it is also possible to determine the required sample size to achieve a predefined accuracy rate. But, due to the lack of space we will leave this simple calculation for the full version of the current paper.

4.2 Results for attack 4 = Trace-driven Attack against the BTB (Asynchronous Attack)

To practically test attack 4, which is also an asynchronous attack, we used a very similar experimental setup as described above to realize attack 2. But, instead of a dummy process that blindly evicts the BTB entries, we used a real spy function. The spy-process evicted the BTB entries by executing conditional branches just like the dummy process. Additionally, it also measured the execution time of these conditional branches. More precisely, it only evicted the entries in the BTB-set that contains the target address of the RSA branch under attack and reported the timing measurements. In this experiment we examined the execution of the conditional branch in the exponentiation routine and not the extra reduction steps of Montgomery Multiplication.

We implemented the spy function in such a way that it only checks the BTB at the beginning or early stages of each montgomery multiplication. Thus, we get exactly one timing measurement per exponentiation step, i.e., multiplication or squaring. Therefore, we could achieve a relatively "clean" measurement procedure. We ran our spy and the cipher process N many times, where N is the sample size. Then we averaged the timing results taken from our spy to decrease the noise amplitude in the measurements. The resulting graph shown in Figure 5 presents our first results for different values of N — clearly visualizing the difference between squaring and multiplication.

As said above one can deduce very clearly from Figure 5 that there is a stabalizing significant cycle difference between multiplication and squaring steps during the exponentiation. Now, that we have verified this BPU-related gap between the successive multiplication and squaring steps during the exponentiation, we want to show now, how simple it is to retreive the secret key with this attack principle 4. To do this, we simply zoom into the following Figure 6 with $N = 10000$ measurements. This yields then the picture on the right side, showing the exponent bits 89 to 104 for $N = 10000$ measurements.

We would like to remark that the sample size of 10000 measurements might appear quite high at first sight. But using some more sophisticated tricks (which are out of the scope of the present paper) we could have a meaningful square/multiply cycle gap using only a few measurements.

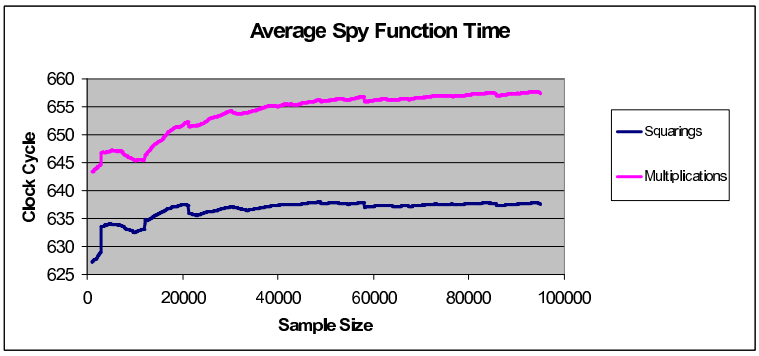

Fig. 5. Increasing gap between multiplication and squaring steps due to missing BTB entries.

5 Conclusions and recommendations for further research

Along the theme of the recent research efforts to explore software side-channels attacks against commodity PC platforms, this paper has identified the branch prediction capability of modern microprocessors as a new security risk which has not been known so far. Using RSA, the most popular public encryption/signature scheme and its most popular open source implementation, openSSL, we have shown that there are various attack scenarios how an attacker could exploit a CPU's branch prediction unit. Also, we have successfully implemented a very powerful attack (Attack 4 = Trace-driven Attack against the BTB which even has the power to break prominent side-channel security mechanisms like those proposed by [JY,Wal]. The practical results from our experiments should be encouraging to think about efficient and secure software mitigations for this kind of new side-channel attacks. As an interesting countermeasure the following branch-less exponentiation method, also known as "atomicity" from [CCJ] comes to our mind. More countermeasures to thwart software-implementations of RSA against branch-prediction attacks will be finally presented in the full version of this extended abstract.

Similarly to other very recent software side-channel attacks against RSA and AES, cf. [Per,OST06] our practically simplest attacks rely on a CPU's Simultaneous Multi Threading (SMT) capability, cf. [Sha]. While SMT seems at first sight a necessary requirement of our asynchronous attacks, we strongly believe that this is just a matter of clever and deeper system's programming capabilities and that this requirement could be removed along some ideas as mentioned in [Hu,OST06]. Thus, we think it is of highest importance to repeat our asynchronous branch prediction attacks also on non-SMT capable CPU's.

References

[Ber] D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005. Available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[BB] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Proceedings of the* 12^{th} *Usenix Security Symposium*, pages 1-14, 2003.

[CEPW] Y. Chen, P. England, M. Peinado, and B. Willman. High Assurance Computing on Open Hardware Architectures. Technical Report, MSR-TR-2003-20, 17 pages, Microsoft Corporation, March 2003. Available at: ftp://ftp.research.microsoft.com/pub/tr/tr-2003-20.ps

[CCJ] B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: side-channel atomicity. *IEEE Transactions on Computers*, volume 53, issue 6, pages 760-768, June 2004.

[CNK] J.-S. Coron, D. Naccache, and P. Kocher. Statistics and Secret Leakage. *ACM Transactions on Embedded Computing Systems*, volume 3, issue 3, pages 492-508, August 2004.

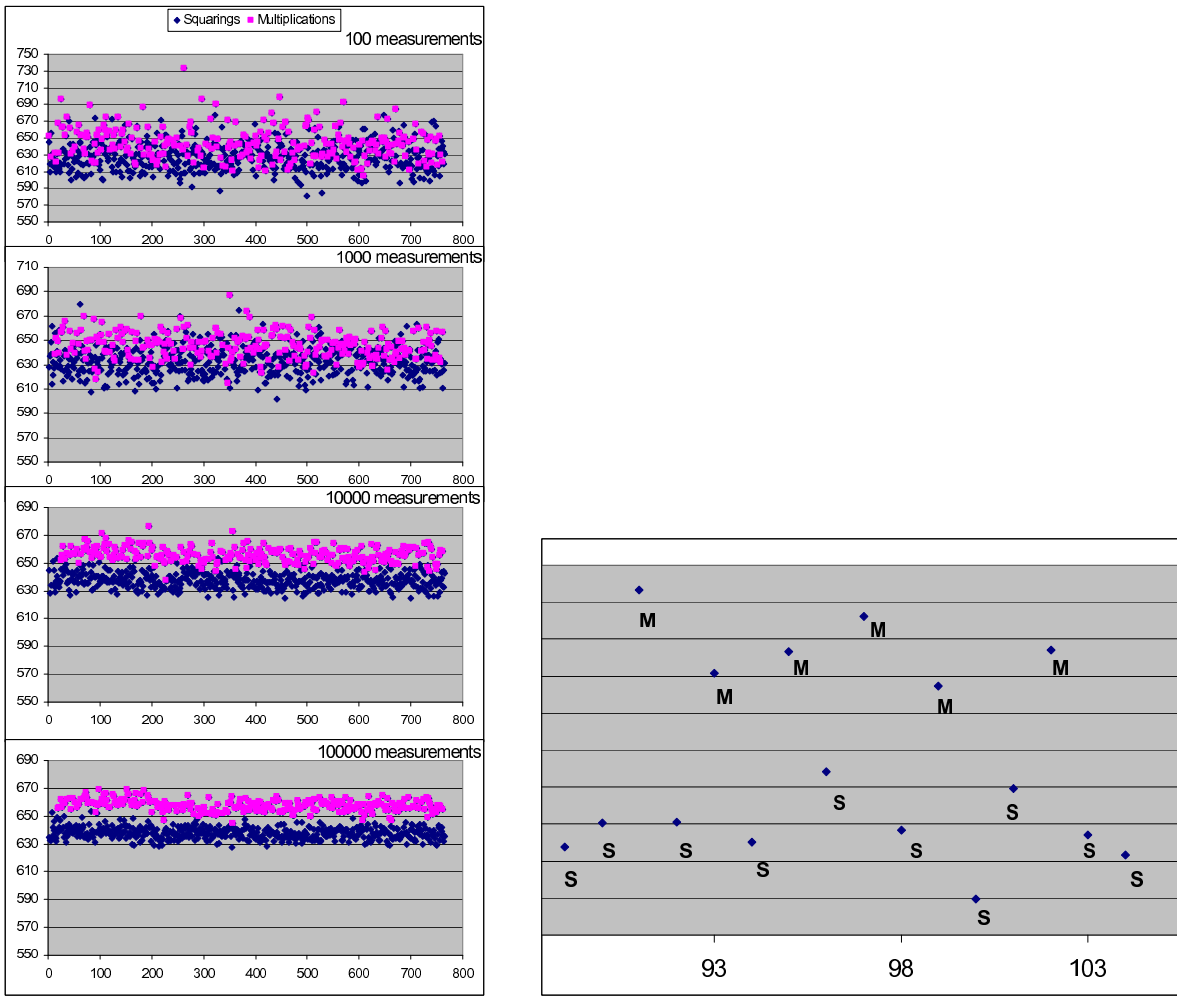

Fig. 6. Connecting the spy-induced BTB misses and the square/multiply cycle gap.

[DKLMQ] J. F. Dhem, F. Koeune, P. A. Leroux, P. Mestre, J.-J. Quisquater, and J. L. Willems. A Practical Implementation of the Timing Attack. *Proceedings of the 3rd Working Conference on Smart Card Research and Advanced Applications - CARDIS 1998*, J.-J. Quisquater and B. Schneier, editors, Springer-Verlag, LNCS vol. 1820, January 1998.

[ELMP$^+$] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *IEEE Computer*, volume 36, issue 7, pages 55-62, July 2003.

[GRAB] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, volume 7, issue 2, May 2003.

[Gra] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*, Intel Press, 2006.

[Hu] W. M. Hu. Lattice scheduling and covert channels. *Proceedings of IEEE Symposium on Security and Privacy*, IEEE Press, pages 52-61, 1992.

[JY] M. Joye and S.-M. Yen. The Montgomery powering ladder. *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski Jr, Ç. K. Koç, and C. Paar, editors, pages 291-302, Springer-Verlag, LNCS vol. 2523, 2003.

[Koc] P. C. Kocher. Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology - CRYPTO '96*, N. Koblitz, editors, pages 104-113, Springer-Verlag,

LNCS vol. 1109, 1996.

[MvOV] A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, New York, 1997.

[MMK] M. Milenkovic, A. Milenkovic, and J. Kulick. Microbenchmarks for Determining Branch Predictor Organization. *Software Practice & Experience*, volume 34, issue 5, pages 465-487, April 2004.

[open] Openssl: the open-source toolkit for ssl / tls. Available online at http://www.openssl.org/.

[OST05] D. A. Osvik, A. Shamir, and E. Tromer. Other People's Cache: Hyper Attacks on HyperThreaded Processors. Presentation available at: `http://www.wisdom.weizmann.ac.il/~tromer/`.

[OST06] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, D. Pointcheval, editor, pages 1-20, Springer-Verlag, LNCS vol. 3860, 2006.

[PH] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* 3rd edition, Morgan Kaufmann, 2005.

[Pea] S. Pearson. *Trusted Computing Platforms: TCPA Technology in Context*, Prentice Hall PTR, 2002.

[Per] C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005. Available at: http://www.daemonology.net/hyperthreading-considered-harmful/

[PL] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing*, 3rd edition, Prentice Hall PTR, 2002.

[Sch] W. Schindler. A Timing Attack against RSA with the Chinese Remainder Theorem. *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. Koç and C. Paar, editors, pages 109-124, Springer-Verlag, LNCS vol. 1965, 2000.

[Sha] T. Shanley. *The Unabridged Pentium 4 : IA32 Processor Genealogy.* Addison-Wesley Professional, 2004.

[She] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* McGraw-Hill, 2005.

[SPL] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. *IEEE Symposium on Security and Privacy*, pages 211-223, 1995.

[Smi1] S. W. Smith. *Trusted Computing Platforms: Design and Applications*, Springer-Verlag, 2004.

[TCG] Trusted Computing Group, `http://www.trustedcomputinggroup.org`.

[UNRS$^+$] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, L. Smith. Intel Virtualization Technology, *IEEE Computer*, volume 38, number 5, pages 48-56, May 2005.

[Wal] C. D. Walter. Montgomery Exponentiation Needs No Final Subtractions. *IEE Electronics Letters*, volume 35, number 21, pages 1831-1832, October 1999.