On the Power of Simple Branch Prediction Analysis

Onur Acıiçmez
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331, USA
aciicmez@eecs.oregonstate.edu

Çetin Kaya Koç
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331, USA
and
Information Security Research Center
Istanbul Commerce University
Eminönü, Istanbul 34112, Turkey
koc@cryptocode.net

Jean-Pierre Seifert
Applied Security Research Group
Faculty of Science and Science Education
University of Haifa,
Haifa 31905, Israel
and
Institute for Computer Science
University of Innsbruck
6020 Innsbruck,
Austria
jeanpierreseifert@yahoo.com

ABSTRACT

Very recently, a new software side-channel attack, called Branch Prediction Analysis (BPA) attack, has been discovered and also demonstrated to be practically feasible on popular commodity PC platforms. While the above recent attack still had the flavor of a classical timing attack against RSA, where one uses many execution-time measurements under the same key in order to statistically amplify some small but key-dependent timing differences, we dramatically improve upon the former result. We prove that a carefully written spy-process running simultaneously with an RSA-process, is able to collect during one *single* RSA signing execution almost all of the secret key bits. We call such an attack, analyzing the CPU's Branch Predictor states through spying on a single quasi-parallel computation process, a *Simple Branch Prediction Analysis (SBPA)* attack — sharply differentiating it from those one relying on statistical methods and requiring many computation measurements under the same key. The successful extraction of almost all secret key bits by our SBPA attack against an openSSL RSA implementation proves that the often recommended blinding or so called randomization techniques to protect RSA against side-channel attacks are, in the context of SBPA attacks, totally useless. Additional to that very crucial security implication, targeted at such implementations which are assumed to be at least statistically secure, our successful SBPA attack also bears another equally critical security implication. Namely, in the context of simple side-channel attacks, it is widely believed that equally balancing the operations after branches is a secure countermeasure against such simple attacks. Unfortunately, this is not true, as even such "balanced branch" implementations can be completely broken by our SBPA attacks. Moreover, despite sophisticated hardware-assisted partitioning methods such as memory protection, sandboxing or even virtualization, SBPA attacks empower an unprivileged process to successfully attack other processes running in parallel on the same processor. Thus, we conclude that SBPA attacks are much more dangerous than previously anticipated, as they obviously do not belong to the same category as pure timing attacks.

ASIACCS'07, March 20-22, 2007, Singapore.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: [Public key cryptosystems, Code breaking]

General Terms

Security

Keywords

Branch Prediction Analysis, Modular Exponentiation, RSA, Side Channel Analysis

1. INTRODUCTION

Deep CPU pipelines paired with the CPU's ability to fetch and issue multiple instructions at every machine cycle led to the concept of superscalar processors. Superscalar processors admit a theoretical or best-case performance of less than 1 machine cycle per completed instructions, cf. [36]. However, the inevitably required branch instructions in the underlying machine languages were very soon recognized as one of the most painful performance killers of superscalar processors. Not surprisingly, CPU architects quickly invented the concept of branch predictors in order to circumvent those performance bottlenecks. Thus, it is not surprising that there has been a vibrant and very practical research on more and more sophisticated branch prediction mechanisms, cf. [28, 34, 36]. Unfortunately, a very recent paper, cf. [1], identified branch prediction as a novel and unforeseen side-channel, thus being another new security threat within the computer security field. 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. [6, 18]. 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. [34]. These factors create noise, and therefore it is much more difficult 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. [3], who demonstrated a remote timing attack over a real local network. They simply adapted the attack principle as introduced in [33] to show that the RSA implementation of OpenSSL [25] — 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. [2, 26, 27, 30], which comes in different forms. Although the cache itself has been known for a long time being a crucial security risk of modern CPU's, cf. [37, 14], the above papers were the first proving such vulnerabilities practically and raised large public interest in such vulnerabilities.

Especially in the light of ongoing Trusted Computing efforts, cf. [39], which promise to turn the commodity PC platform into a trustworthy platform, cf. also [4, 8, 10, 29, 39, 40], the formerly described side channel attacks against PC platforms are of particular interest. This is due to the fact that side channel attacks have been completely ignored by the Trusted Computing community so far. 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 [10, 40] 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. [31, 10].

Following this interesting new research vector, Acıiçmez, Koç, and Seifert, cf. [1], just recently discovered that the branch prediction capability, common to all modern high-end CPU's, is another new side-channel posing a novel and unforeseen security risk. The authors presented different branch prediction attacks on simple RSA implementations as a case study to describe the basics of their novel attacks an adversary can use to compromise the security of a platform. In order to do so, they gradually developed from an obvious attack principle more and more sophisticated attack principles, resulting in four different scenarios. To demonstrate the applicability of their attacks they complemented their scenarios by showing the results of selected practical implementations of their various attack scenarios.

Irrespectively of their achievements, it is obvious that all of their attacks still had the flavor of a classical timing attack against RSA. Indeed, careful examination of these four attacks introduced in [1] shows that they all require many measurements to finally reveal the secret key. In a timing attack, the key is obtained by taking many execution time-measurements under the same key in order to statistically amplify some small but key-dependent timing differences, cf. [18, 7, 33]. Thus, simply eliminating the deterministic time-dependency of the RSA signing process of the underlying key by very well understood and also computationally cheap methods like message blinding or secret exponent masking, cf. [18], such statistical attacks are easy to mitigate. Therefore, it is quite natural that timing attacks caused no real threat to the security of PC platforms.

Unfortunately, the present paper teaches us that this "let's think positive and relax" assumption is quite wrong! Namely, we dramatically improve upon the former result of [1] in the following sense. We prove that a carefully written spy-process running simultaneously with an RSA-process is able to collect during one *single* RSA signing execution almost all of the secret key bits. We call such an attack, analyzing the CPU's Branch Predictor states through spying on a single quasi-parallel computation process, a *Simple Branch Prediction Analysis (SBPA)* attack. In order to clearly differentiate those branch prediction attacks that rely on statistical methods and require many computation measurements under the same key, we will call those *Differential Branch Prediction Analysis (DBPA)* attacks. However, additional to that very crucial security implication — SBPA is able to break even such implementations which are assumed to be at least statistically secure — our successful SBPA attack also bears another equally critical security implication. Namely, in the context of simple side-channel attacks, it is widely believed that equally balancing the operations after branches is a secure countermeasure against such simple attacks, cf. [16]. Unfortunately, this is not true, as even such "balanced branch" implementations can be completely broken by our SBPA attacks.

The present paper is organized as follows. The next section gives some background information including the structure and functionality of a general Branch Prediction Unit and some important details to understand the rest of the paper. Also, it gives a brief overview about RSA, a short primer on its usual practical implementation, and some definitions used throughout the paper. Section 3 briefly recalls the so called "Trace-driven Attack against the BTB" due to Acıiçmez, Koç, and Seifert, cf. [1]. Hereafter, we turn to our enhanced analysis of a trace-driven attack against a Branch Target Buffer (BTB). The positive results of our practical experiments, yielding a practical SBPA attack against the RSA implementation of OpenSSL are presented in section 5. Finally, we summarize the results of the present paper, conclude with some remarks and outline some future research in the last section.

2. BACKGROUND, DEFINITIONS AND PRELIMINARIES

2.1 Branch Prediction Unit

Superscalar processors have to execute instructions specu-

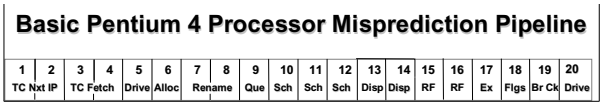

Figure 1: 20 stage Misprediction Pipeline.

latively to overcome control hazards, cf. [36]. 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 their efficiency is thus simply 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. [36].

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 the branch 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. The following Figure 1 shows the so called "20 stage Misprediction Pipeline" of the famous Intel Pentium 4 Processor. Thus, in order to address such branch bottlenecks and 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. [36].

- *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)*.

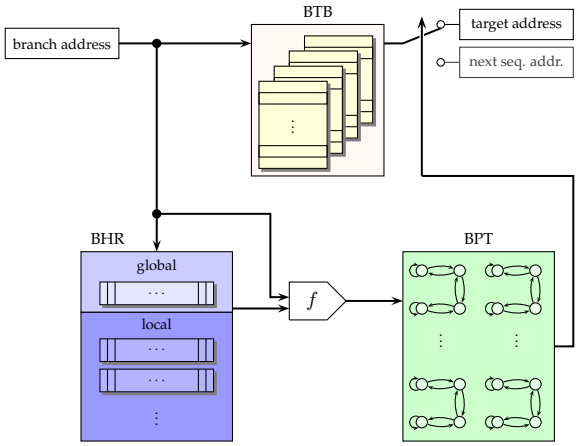

Figure 2: Branch Prediction Unit Architecture.

Overall common to all *Branch Prediction Units (BPU)* is the following rough Figure 2. 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 some 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 BTB is functional and architectural very similar to an ordinary cache, and indeed used as a cache for previously seen branch target adresses of "cached" branch instructions at certain adresses.

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, etc., cf. [36].

2.2 RSA and the Binary Square-and-Multiply Exponentiation Algorithm

RSA is the most widely used public key cryptosystem which was developed by Rivest, Shamir and Adleman, cf. [20]. 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 secret, and N is the public modulus. Here, N is the 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, from which 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 (even on todays very powerful PC platforms) still very expensive in terms of the execution time. Therefore, actual implementations of RSA usually employ very efficient and optimized algorithms to accelerate the result of this operation. However, tailored to our "proof-of-concept" purpose, we explain

```
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)
return S
```

Figure 3: Binary version of Square-and-Multiply Exponentiation Algorithm

```
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)
    else [if d_i = 1] then
        R_0 = R_0 ∗ R_1 (mod N)
        R_1 = R_1 ∗ R_1 (mod N)
return R_0
```

Figure 4: Balanced Montgomery Powering Ladder

in the next subsection only the most widely and simplest used algorithm.

2.2.1 Binary Square-and-Multiply Exponentiation Algorithm.

The binary version of the classical Square-and-Multiply Algorithm (SM) is the simplest way to perform an exponentiation. We want to compute $M^d (\text{mod } N)$, where d is an n-bit number, i.e., $d = (d_0, d_1, ..., d_{n-1})_2$. Figure 3 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 on RSA. In an efficient RSA implementation all of the multiplications and squarings are then actually performed using a special modular multiplication algorithm, so called Montgomery Multiplication, cf. [20]. But contrary to many other papers, cf. [3, 7, 33], whose target is this special Montgomery Multipliction, we are (to keep things in this paper as simple as possible) only interested in this SM branch.

2.2.2 Balanced Montgomery Powering Ladder

In the context of side channel attacks, cf. [6, 18, 16], it was quickly "agreed" that simple side-channel attacks could be (simply) mitigated by avoiding the unbalanced and key-dependent conditional branch in the above Figure 3, and just insert dummy operations into the flow in order to make the operations after the conditional branch more balanced, cf. [16]. As this "dummy equipped" binary SM algorithm still had some negative side-effects, cf. [16], a very active research area arose around the so called Balanced Montgomery Powering Ladder, as shown in the Figure 4.

This exponentiation is assumed to be "intrinsically secure" against simple side-channel attacks, cf. [16], and also has many compuatational advantages over the above basic SM algorithm. Unfortunately, we will explain and see later, that all those "balanced branch" exponentiation algorithms are "intrinsically insecure" in the presence of SBPA attacks.

2.3 Multi-Threading, spy and crypto processes

With the advent of the papers from [26, 27, 30] a new and very interesting attack paradigm was initiated. This relies on the massive multi-threading (quasi-parallel) capabilities of modern CPU's, whether hardware-managed or OS-managed, cf. [22]. While purely single-threaded processors run threads/processes clearly serial, the OS manages to execute several programs in a quasi parallel way, cf. [35]. The OS basically decomposes an application into a series of short threads that are ordered with other application threads. On the other side, there are also certain processors, so called hardware-assisted multi-threaded CPU's, which enable a much finer-grained quasi-parallel execution of threads, cf. [36, 34]. Here, some "cheap" CPU resources are explicitly doubled (tripled, etc.), while some others are temporarily shared. It allows them to have two or many other processes running *quasi* parallel on the same processor, as if there were two or more logical processors [36, 35]. This allows then indeed a fine-grained instruction-level multi-threading, cf. [36].

Irrespectively of single-threaded or hardware-assisted multi-threaded, some logical elements are always shared, which enables one process to *spy* on another process, as the shared CPU elements leak some so called metadata, cf. [26, 27]. Of course, the sharing of the resources does not allow a direct reading of the other applications data, as the memory protection unit (MMU or Virtual Machine) stricly enforces an application memory separation. One such example of a shared element, which is the central point of interest for this paper is the highly complex BPU of modern CPU's.

The new paradigm put forward by [26, 27, 30], although already implicitly pointed out by Hu [14], consists of quasi-parallel processes, called *spy* process and *crypto* process. As the name suggest, the spy process tries to infer some secret data from the parallel executed crypto process by observing the leaked metadata. In the most extreme and most practical scenario, both processes run completely independently of each other, and this scenario was termed asynchronous attack by [27].

Given the very complex process structures and their handling by a modern OS, cf. [35], the following heuristic is quite obvious.

> *A hardware-assisted multi-threading CPU will simplify a successfull spy process, as:*
>
> 1. *Some inevitable "noise" due to the respective thread switches will be absorbed by the CPU's hardware-assistance.*
> 2. *The instruction-level threading capability enhances the time-resolution of the spy-process.*

In the other case, one needs a very sophisticated OS expertise and a deep thread scheduling expertise, cf. [22]. As the above paradigm and all its subtle implementation details heavily depend on the underlying OS, CPU type and frequency, etc. we will not deepen further those technical details here, and just assume the existence of a suited *spy* process and a corresponding *crypto* process in a hardware-assisted multi-threading environment.

3. RECALLING ATTACK 4 FROM [ASK]: TRACE-DRIVEN ATTACK AGAINST THE BTB

The attack 4 from Acıiçmez, Koç, and Seifert, cf. [1], also known as "Trace-driven Attack against the BTB", utilizes the above outlined quasi-parallel paradigm of a spy and crypto process running on the same platform. It works as follows.

A "protected" crypto process executes the RSA signing process by using one of the above exponentiation algorithms, i.e. 3 or 4 or any other exponentiation algorithm involving a branch depending on the secret key bits, and therefore executes a sequence of conditional branches, as long as the size of the secret key d. Also a spy process is executed simultaneously with the cipher and it continuously does the following:

1. continuously executes a number of branches, and
2. measures the overall execution time of all its branches

in such a way that all of these branches map to the same BTB set which also stores the specific conditional branch determined by the secret key bits of the crypto process. This requires that the number of branches in the spy process needs to be equal to the associativity of the underlying BTB, i.e., to its number of ways. Recall that it is easy to understand the properties of the BTB using simple benchmarks as explained in [21].

Let's analyze what's happening if the adversary starts the spy process before the cipher. It simply means that when the cipher starts the encryption (= signing), the CPU *cannot* find the target address of the target branch in the BTB and the prediction *must* be *not-taken*, cf. [36]. Furthermore, we can distinguish two cases depending on the currently processed secret key bit:

- 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 its branches, it will encounter a misprediction on the branch that has just been evicted. As the spy-process also measures the execution time of all its branches, it can simply detect whenever the cipher modified the BTB, meaning that the execution time of these spy branches takes a little longer than usual.
- If the branch turns out to be not taken, then no misprediction will occur and the BTB does not need to be updated. When the spy-process re-executes its branches, measures the execution time of all its branches, it can simply infer that the cipher had not modified the BTB, and the target branch was not taken by the crypto process.

Thus, the adversary can simply determine the complete execution flow of the cipher process by continuously performing the above very simple spy strategy, i.e., just executing spy branches and measuring their overall execution time. Therefore, the spy process will see the complete prediction/misprediction trace of the target branch, and is able to infer the secret key. Following [27], this kind of attack was named an asynchronous attack, as the adversary-process needs *no* synchronization at all with the simultaneous crypto process — it is just following his own paradigm: continuously execute spy branches and measure their overall execution time.

In order to demonstrate the feasibility of the above attack [1] had run the RSA process in the presence of a spy process under the same secret key N times, and averaged then the timing results taken from their spy to decrease the noise amplitude in the measurements. Their resulting graphs are shown for different values of N in Figure 5 — clearly visualizing the stabilizing effect in the cycle gap between squaring and multiplication.

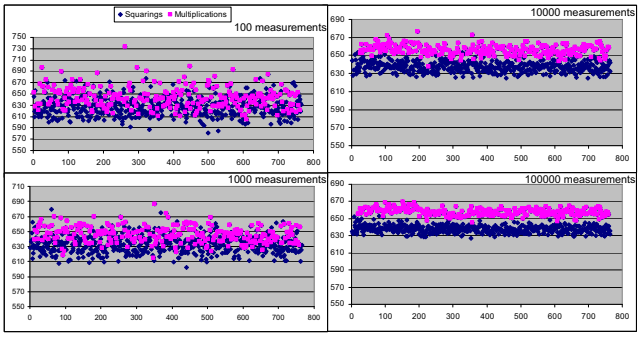

Figure 5: Stabilizing square/multiply cycle gap resulting from the above trace-driven attack against the BTB

4. IMPROVING TRACE-DRIVEN ATTACKS AGAINST THE BTB

In this section, we present our improvement over the DBPA attack from [1], which we outlined in the last section. However, in order to logically derive our final successful SBPA result against some version of the binary square and multiply exponentiation for RSA, we have to investigate the situation a bit deeper.

If we consider the above Figure 5, we can certainly draw the conclusion that from spy processes like this, there is no hope for a successful SBPA. At first sight this looks quite astonishing for the following reason. In a certain sense, the trace-driven attack against the BTB from [1] is very similar to the cache eviction attacks of [30, 27, 22]. In these attacks, a spy process is also continuously testing through timing measurements which of its private data had been evicted by the crypto process. And especially in the RSA OpenSSL 9.7 case from [30], the measurement quality was high enough to get lots of secret key bits by spying on one single exponentiation, i.e., inferring by simple time measurements which data the crypto process had loaded into the data cache, to perform the RSA signing operation.

However, there is one fundamental difference, setting BPA attacks apart from pure data cache eviction attacks. Attacking the BTB, although being itself acting like a simple cache, is actually targeting the instruction flow, which is magnitudes more complicated than the data flow within the memory hierarchy, i.e., between the L1 data cache and the main memory. Therefore, as already mentioned in the above section 2.1, numerous architectural enhancements take care

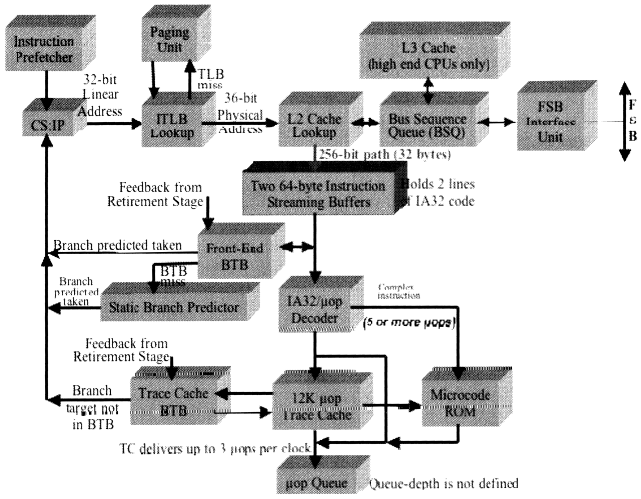

Figure 6: Front-End Instruction Pipeline Stages feeding the μop Queue

that a deeply pipelined superscalar CPU like Pentium 4 cannot get too easily stalled by a BTB miss. When considering just (and what is publicly known) the Front-End Instruction Pipeline Stages between the Instruction Prefetching Unit and the resulting feeding into the so called μop Queue, as shown in the Figure 6 below, we recognize that only this Front-End Instruction path is much more complicated than the data flow path, cf. [34].

If we inspect the above Figure 6 in more depth, we can recognize that the Pentium 4 has two different BTB's: a Front-End BTB and a Trace-Cache BTB. As the architectural reasons for this second Trace-cache BTB are out of interest for this paper, we refer the interested readers to citeShen,TraceCache. However, more interesting is the information on their sizes, and especially their joint functionality which we can partially learn from [34, pp. 913-914]: *The travels of a conditional branch instruction*. The Front-End BTB has a size of 4096 entries, whereas the Trace-Cache BTB has only a size of 512 entries, i.e., the Front-End BTB is a superset of the Trace-Cache BTB.

The most interesting fact that we can draw from this doubled BTB is the following. Executing a certain sequence of branches in the spy process which evicts just the Front-End BTB might not necessarily suffice to completely enforce the CPU *not to* find the target address of the target branch in some of the BTB's. A certain hidden interaction between Front-End BTB and Trace-Cache BTB might allow for some "short-term" victim address evictions, but still store the target branch in one of the BTB's .

Thus, we decided to let the spy process continuously do the following. Continuously execute a certain fixed sequence of, say t, branches to evict the target branch's entry out of the BTBs and measure the overall execution time of all these branches. This is exactly what is done in the earlier attack of Acıiçmez et al. except for a single difference, which transforms their trace-driven attack from a DBPA attack into an extremely powerful SBPA attack. The *optimal* number of t branches turns out to be significantly larger than the number of associativity, which is the exact value used in [1]. The increased value for t guarantees the eviction of the target entry from all different places that can store it, e.g., from both Front-End BTB and Trace-Cache BTB.

The value of t also affects the cycle gap between squaring and multiplication in the following way. As mentioned in the previous section, when the target branch is evicted from the BTB and 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 its branches, it will encounter a misprediction on the branch that has just been evicted.

A fact that was not mentioned above is that this misprediction will also trigger further mispredictions since the entry of the evicted spy branch needs to be re-stored and another not-yet-reexecuted spy branch entry has to be evicted, which will also cause other mispredictions. At the end, the execution time of this spy step is expected to suffer from *many* misprediction delays resulting in a very high gap between squaring and multiplication. However, this scenario only works out if the entries are completely evicted from all possible locations. As can be seen in Figure 5, the gap here is only 20 cycles, which indicates that the above scenario is not valid for this particular attack, i.e., for this value of t. Increasing t to its optimal value also enforces our scenario and guarantees a very large gap composed of several misprediction delays. This fact is clear, when considering the gap around 1000 cycles in our SBPA attack, i.e., our improved trace-driven attack.

The optimal value of t is eventually machine dependent and (most likely) also depends on the particular set of software, i.e., the OS running on the machine. Therefore, an adversary needs to carefully tune, e.g., empirically determine the optimal value for t, the spy process on the attacked machine.

5. PRACTICAL RESULTS

To validate our aforementioned enhanced "BTB eviction strategy", we performed some practical experiments. As usual in this context, we have chosen to carry out our experimental attacks in a popular simultaneous multithreading environment, cf. [34], as this CPU type simplifies the context switching between the spy and the crypto process. In our above outlined setting, the adversary can apply this asynchronous attack without any knowledge on the details of the used branch prediction algorithm or any deeper BTB structure knowledge.

As in [1], we performed this attack on a very simple RSA implementation that employed a square-and-multiply exponentiation and also Montgomery multiplication *with* dummy reduction. We used the RSA implementation from 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, removed the CRT mode, and added the dummy reduction step. We used random plaintexts generated by the `rand()` and `srand()` functions, as available in the standard C library, and measured the execution time in terms of clock cycles using the cycle counter instruction `RDTSC`, which is available in user-level.

Our experimental results of this enhanced "BTB eviction strategy" for RSA-sign with a 512 bit key-length are shown in the following Figure 7.

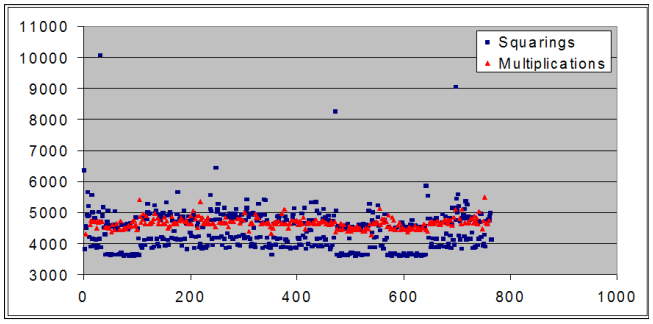

Figure 7: Results of SBPA with an improved resolution.

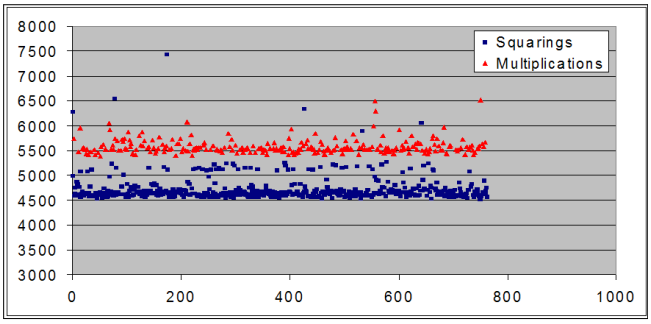

Figure 9: Best result of our SBPA against openSSL RSA, yielding 508 out of 512 secret key bits.

5.1 A simple heuristic to enhance the measurement resolution

As recognizable from the above Figure 7, our repeated spy-execution of a certain fixed sequence of branches certainly enhanced the resolution for *one single* RSA-sign measurement. Indeed, comparing Figure 7 with Figure 5 one could say that this simple trick "saved" an averaging of about 1000 to 10000 different measurements.

Although this is already a clear dramatic resolution enhancement, we weren't able to directly amplify this SBPA resolution anymore. Being stuck with this resolution, and getting not enough secret key bits, we employed another more heuristic, but very powerful argument based on the following fact.

On an average PC (client or server) running Windows, Linux, etc. there are many quasi-parallel processes running, whether system-processes or user-initiated processes. The time when such processes are running can be assumed to be random and it heavily influences the timing-behavior of every other process, for e.g., our spy and crypto process.

Therefore, there is a statistical chance to perform some of our measurements during a timeframe when such influences are minimal, which leads us to our following heuristic:

> there must exist among all those measurements also some quite "clear" measurements.

We call this argument the *time-dependent random self-improvement* heuristic. Applying this heuristic simply means that we just have to do some SBPA measurements, say at several independent times, and we can be sure that among those there will be at least "one unusually good" individual measurement, which will be our final SBPA. To validate this heuristic, we performed then ten different "random SBPA attacks on the same 512 bit key, from which we show in Figure 8 only 4 very different ones. Without doubt, they are all quite different although they process the same key, thus supporting our heuristic quite well.

And indeed, the following experimental result, also being among those ten measurements, clearly shows that there is one exceptionally clear one, which directly reveals 508 out of 512 secret key bits.

Armed with this final experimental result, we safely can claim that we have lifted the work of [1] to the much more powerful SBPA area.

6. CONCLUSIONS

Branch Prediction Analysis (BPA), which recently led to a new software side-channel attack, still had the flavor of classical timing attacks against RSA. Timing attacks use many execution-time measurements under the same key in order to statistically amplify some small but key-dependent timing differences. In this paper, we have dramatically improved the former results of [1] and showed that a carefully written spy-process running simultaneously with an RSA-process, is able to collect during one *single* RSA signing execution almost all of the secret key bits. We call this attack, analyzing the CPU's Branch Predictor states through spying on a single quasi-parallel computation process, a *Simple Branch Prediction Analysis (SBPA)* attack — sharply differentiating it from those one relying on statistical methods and requiring many computation measurements under the same key. The successful extraction of almost all secret key bits by our SBPA attack against an openSSL RSA implementation proves that the often recommended blinding or so called randomization techniques to protect RSA against side-channel attacks are, in the context of SBPA attacks, totally useless. Additional to that very crucial security implication, targeted at such implementations which are assumed to be at least statistically secure, our successful SBPA attack also bears another equally critical security implication. Namely, in the context of simple side-channel attacks, it is widely believed that equally balancing the operations after branches is a secure countermeasure against such simple attacks. Unfortunately, this is not true, as even such "balanced branch" implementations can be completely broken by our SBPA attacks. Moreover, despite sophisticated hardware-assisted partitioning methods such as memory protection, sandboxing or even virtualization, SBPA attacks empower an unprivileged process to successfully attack other processes running in parallel on the same processor. Thus, we conclude that SBPA attacks are much more dangerous than previously anticipated, as they obviously do not belong to the same category as pure timing attacks.

More importantly, since our new attack requires only one single execution observation, and thus significantly differs from the earlier timing attacks, the SBPA discovery opens new and very interesting application areas. It especially endangers those crptographic/algorithmic primitives, whose nature is an intrinsic and input dependent branching process. Here, we especially target the modular reduction and the modular inversion part. In practical implementations of popular cryptosystems they are often used in such cases,

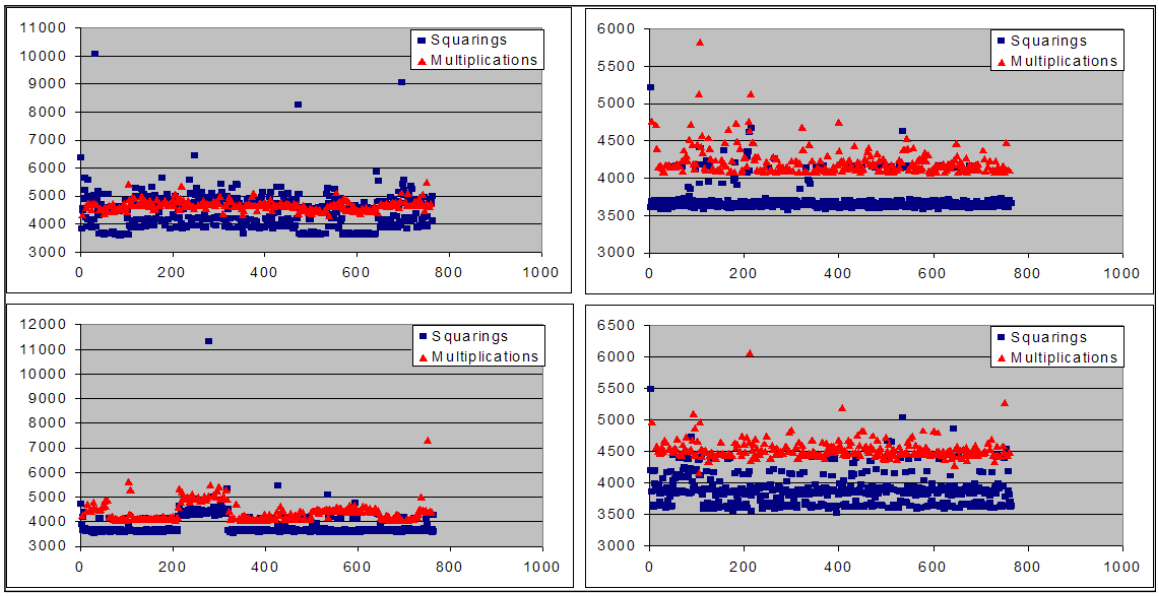

Figure 8: Enhancing a bad resolution via independent repetition.

where one parameter of the respective algorithm (i.e., modular reduction or modular inversion) is an important secret parameter of the underlying cryptosystem. Let us briefly mention a few but important situations for reduction and inversion, where a successfull SBPA attack could lead to a serious security compromise.

- Modular reduction (mod p and mod q) is used in the initial normalization process of RSA when using the Chinese Remainder Theorem, cf. [20]. And indeed, [15, 17] already pointed out that the classical pencil and paper division algorithm could leak through certain side channels the secret knowledge of p and q.

- Inversion is also very often used as a statistical side channel atttack countermeasure to blind messages during RSA signature computations, cf. [18, 33], thus effectively combating classical timing attacks, cf. [3].

- Inversion is the main ingredient during the RSA key generation set-up to compute the secret exponent from the public exponent and the totient function of the respective RSA modulus.

- Inversion is also used in the (EC)DSA, cf. [20], and just the leakage of a few secret bits of the respective ephemeral keys, cf. [11, 23, 24], leads to a total break of the (EC)DSA.

Classical timing attacks *cannot* compromise such operations solely because they rely on capturing *many* measurements and statistical analysis with the same input paramters, whereas the above situations execute the reduction or inversion part only *once* for a specific input set. We feel that our findings will eventually result in a serious revision of current software for various public-key cryptosystem implementations, and that there will arise a new research vector along our results.

7. REFERENCES

[1] O. Acıiçmez, J.-P. Seifert, and Ç. K. Koç. Predicting secret keys via branch prediction. Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, to appear.

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

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

[4] 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`

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] N. A. Howgrave-Graham and N. P. Smart. Lattice Attacks on Digital Signature Schemes. *Design, Codes and Cryptography*, vol. 23, pp. 283290, 2001.

[12] J. Handy. *The cache memory book (2nd ed.): the authoritative reference on cache design*, Academic Press, Inc., Orlando, FL, USA, 1998.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, volume 5, issue 1, Feb. 2001.

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

[15] M. Joye, and K. Villegas. A protected division algorithm. In P. Honeyman, Ed., Smart Card Research and Advanced Applications (CARDIS 2002), pages 69-74, Usenix Association, 2002.

[16] 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.

[17] H. Kahl. SPA-based attack against the modular reduction within a partially secured RSA-CRT implementation. Cryptology ePrint Archive, Report 2004/197, 2004, `http://eprint.iacr.org/197.pdf`.

[18] 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.

[19] R. Mayer-Sommer. Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards. .K. Ko and C. Paar (ed.), Cryptographic Hardware and Embedded Systems - CHES 2000, Springer-Verlag, 2000, LNCS vol. 1965, pp. 78-92.

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

[21] 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.

[22] M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. Proceedings of Selected Area of Cryptology (SAC 2006), Montreal, Canada, August 2006, to appear at Springer LNCS.

[23] P. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology*, vol. 15, no. 3, pp. 151176, Springer, 2002.

[24] P. Q. Nguyen and I. E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Design, Codes and Cryptography*, vol. 30, pp. 201 217, 2003.

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

[26] 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/`.

[27] 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.

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

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

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

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

[32] E. Rotenberg, S. Benett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. Proceedings of the 29th Annual ACM/IEEE Intl. Symposium on Microarchitecture, pages 24–34, Dec. 1996.

[33] 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.

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

[35] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating system concepts (7th ed.)*. John Wiley and Sons, Inc., USA, 2005.

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

[37] 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.

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

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

[40] 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.

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