New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures

Onur Acıiçmez[1], Shay Gueron[2,3], and Jean-Pierre Seifert[4]

[1] Samsung Information Systems America, San Jose, USA
[2] Department of Mathematics, University of Haifa, Haifa, 31905, ISRAEL
[3] Intel Corporation, IDC, ISRAEL
[4] Institute for Computer Science, University of Innsbruck, 6020 Innsbruck, AUSTRIA
onur.aciicmez@gmail.com, shay@math.haifa.ac.il, jeanpierreseifert@yahoo.com

February 7, 2007

Abstract. Software based side-channel attacks allow an unprivileged spy process to extract secret information from a victim (cryptosystem) process by exploiting some indirect leakage of "side-channel" information. It has been realized that some components of modern computer microarchitectures leak certain side-channel information and can create unforeseen security risks. An example of such MicroArchitectural Side-Channel Analysis is the Cache Attack — a group of attacks that exploit information leaks from cache latencies [4, 7, 13, 15, 17]. Public awareness of Cache Attack vulnerabilities lead software writers of OpenSSL (version 0.9.8a and subsequent versions) to incorporate countermeasures for preventing these attacks.

In this paper, we present a new and yet unforeseen side channel attack that is enabled by the recently published Simple Branch Prediction Analysis (SBPA) which is another type of MicroArchitectural Analysis, cf. [2, 3]. We show that modular inversion — a critical primitive in public key cryptography — is a natural target of SBPA attacks because it typically uses the Binary Extended Euclidean algorithm whose nature is an input-centric sequence of conditional branches. Our results show that SBPA can be used to extract secret parameters during the execution of the Binary Extended Euclidean algorithm. This poses a new potential risk to crypto-applications such as OpenSSL, which already employs Cache Attack countermeasures. Thus, it is necessary to develop new software mitigation techniques for BPA and incorporate them with cache analysis countermeasures in security applications.

To mitigate this new risk in full generality, we apply a security-aware algorithm design methodology and propose some changes to the CRT-RSA algorithm flow. These changes either avoid some of the steps that require modular inversion, or remove the critical information leak from this procedure.

In addition, we also show by example that, independently of the required changes in the algorithms, careful software analysis is also required in order to assure that the software implementation does not inadvertently introduce branches that may expose the application to SBPA attacks.

These offer several simple ways for modifying OpenSSL in order to mitigate Branch Prediction Attacks.

Keywords: Side channel attacks, branch prediction attacks, cache eviction attacks, Binary Extended Euclidean Algorithm, modular inversion, software mitigation methods, OpenSSL, RSA, CRT.

1 Introduction

Side channel attacks are methods by which an attacker can extract secret information from an implementation of a cryptographic algorithm. They come in various flavors, exploiting different security weaknesses of both the cryptographic implementations and the environments on which the cryptographic applications run. MicroArchitectural Side-Channel attacks are a special new class of attacks that exploit the microarchitectural throughput-oriented behaviour of modern processor components. These attacks capitalize on situations where several applications can share the same processor resources, which allows a spy process running in parallel to a victim process to extract critical information.

Cache Attacks are one example of MicroArchitectural Side-Channel attacks. In one flavor of Cache Attacks, the adversary takes advantage of a specifically crafted spy process. This spy first fills the processor's cache before the victim process (i.e., cipher process) takes over at context switch. In the subsequent context switch the spy process experiences cache evictions that were caused by the operation of the victim process, and the eviction patterns can be used for extracting secret information. This concept was used ([17]) for demonstrating an attack on OpenSSL-0.9.7. The attack focused on the Sliding Windows Exponentiation (SWE) algorithm for computing modular exponentiation - part of the RSA decryption phase that uses the server's private key.

As a result, the subsequent version OpenSSL-0.9.8 included mitigations against this attack. The SWE algorithm was replaced with a Fixed Window Exponentiation (FWE) algorithm, using window length 5. In this algorithm, a sequence of 5 repeated modular squares is followed by a modular multiplication by the appropriate power of the base, depending on the value of the 5 scanned bits of the exponent. The powers of the base are computed once, before the actual exponentiation, and stored in a table. Each entry of this table is stored in memory in a way to span multiple cache lines. This way, the eviction of any table entry from the cache does not expose information on the actual the table index of this entry. Another incorporated mitigation is "base blinding": the exponentiation base is multiplied by some factor (unknown to the adversary), and the final exponentiation result is then multiplied by a corresponding factor that cancels out the undesired effect. This technique eliminates the "chosen plaintext" scenario that enables remote timing attacks such as [5, 9].

Branch Prediction Analysis (BPA) and Simple Branch Prediction Analysis (SBPA) attacks are a new type of MicroArchitectural attacks that have been recently published by Acıiçmez et al. [1–3, 6]. These attacks exploit the branch prediction mechanism, which is nowadays a part of all general purpose processors. Microprocessors speed up their performance by using prediction algorithms to guess the most probable code path to execute, and fill the pipeline with the corresponding instructions. When the speculatively executed instruction flow turns out to be wrong, the execution has to start over from the correct one. Good prediction mechanisms/algorithms are those that have high correct prediction rates, and the development of such mechanisms is part of the technological advances in microprocessors. Furthermore, deeper pipelines enhance the average performance. Measurable timing differences between a correct and incorrect prediction are the inevitable outcome of such performance optimization, which is exactly what the BPA/SPBA attacks capitalize on. In theory, a spy process running on the target machine, together with the victim process, can use these timing differences as a side-channel information and deduce the precise execution flow performed by the victim process. This can potentially lead to a complete break of the system if the software implementation of the victim process is written in a way that knowledge of the execution flow provides the attacker with useful information.

Acıiçmez et al. showed that BPA and SBPA can be used to allow a spy process to extract the execution flow of an RSA implementation using the Square and Multiply (S&M) exponentiation algorithm. They demonstrate the results on the S&M algorithm implemented in OpenSSL-0.9.7. The initially published BPA attack required statistical analysis from many runs and could be viewed as difficult to implement in practice. However, the subsequent demonstration of the SBPA attack showed that measurements taken from a *single* run of the S&M exponentiation is sufficient to extract almost all of the RSA secret exponent bits.

The S&M algorithm is not frequently used in practice because there are more efficient exponentiation algorithms such as SWE, which is used by OpenSSL. The attack was mounted on OpenSSL-0.9.7 where the window size was changed from the default value $w = 5$ to $w = 1$ which degenerates SWE to S&M. Attacking the S&M algorithm was only a case study that showed the potential of SBPA. However, as we show in this paper, the actual scope of SBPA attacks is much broader. We identify a novel side-channel attack which is especially enabled by the SBPA methodology.

Obviously, it is unreasonable to handle the new (and also the old) threats by deactivating branch prediction or by disabling multi-process capabilities in general purpose processors and in operating systems. Thus, the conclusion is that cryptographic software applications that run on general platforms need to be (re-)written in an SBPA-aware style. To this end, another focus of this paper is on a software mitigation methodology for protecting applications against SBPA attacks.

The paper is organized as follows. Section 2 briefly recalls some basic facts and details of RSA and of its implementation in OpenSSL. The next section presents then the central contribution of the present paper. This is our novel SBPA side-channel against the BEEA, which enables full reconstruction of the input parameters to the BEEA — even if we assume that they are all completely unknown. Section 4 illustrates new vulnerabilities

which we found in OepnSSL in the presence of potential SBPA attacks. In this section we also explain why these attacks are potentially feasible and can be mounted on almost all platforms. In Section 6 we explain the necessary software countermeasures to protect the openSSL CRT-RSA library against branch prediction attacks. The paper finishes with our conclusions on the BPA story as we currently see it.

2 Preliminaries and Notations

To facilitate smooth reading of the details in the paper, we provide a very brief description of the RSA cryptosystem and its corresponding implementation in OpenSSL. For a more detailed exposition of both topics we refer the reader to [12] and to [16].

2.1 RSA Parameters

RSA key generation starts by generating two large (secret) random primes p and q, where their n-bit long product $N = p\ q$ is used as a public modulus. Then, a public exponent e is chosen ($e = 2^{16} + 1$ by default in OpenSSL), for which $d = e^{-1} \bmod (p-1)(q-1)$ is the private exponent. Figure 1 shows the RSA key generation flow.

RSA is used for both encryption and digital signature, where signing and decrypting messages require the use of the private exponent and modulus, while signature verification and encrypting messages require the used on only the public exponent and modulus. Factoring N (i.e., obtaining p and q) immediately reveals d. If an adversary obtains the secret value d, he can read all of the encrypted messages and impersonate the owner of the key for signatures. Therefore, the main purpose of cryptanalytic attacks on RSA is to reveal either p, q, or d.

input: null
output: $(N, p, q, e, d, d_p, d_q, C_2)$

procedure RSA_KeyGen:

(1) generate random primes p and q
(2) compute $N = p * q$
(3) choose $e \in \mathbb{Z}_N^*$
(4) d = Mod_Inverse(e, $((p-1)(q-1))$) // compute $(d = e^{-1} \bmod (p-1)(q-1))$
(5) (null, d_p) = Division(d, $p-1$) // compute $d_p = d \bmod (p-1)$
(6) (null, d_q) = Division(d, $q-1$) // compute $d_q = d \bmod (q-1)$
(7) C_2 = Mod_Inverse(q, p) // compute $(C_2 = q^{-1} \bmod p)$

Fig. 1. RSA key generation procedure

2.2 RSA Implementation using Chinese Remainder Theorem

OpenSSL, as many other implementations, uses the Chinese Remainder Theorem (CRT). This allows to replace modular exponentiation with full-length modulus N with two modular exponentiations with half-length modulus p and q. This speedups the modular exponentiation by a factor of approximately 4.

In order to employ CRT, the following additional values are computed during the key generation procedure: $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, and $C_2 = q^{-1} \bmod p$. To decrypt a message M, the CRT procedure starts with reducing M modulo p and q, calculating $M_p = M \bmod p$, $M_q = M \bmod q$. Then, the two modular

exponents $S_p = M_p^{d_p} \bmod p$ and $S_q = M_q^{d_q} \bmod q$ are calculated, and the desired result S is obtained bye the so-called "Garner re-combination" $S = S_q + (((S_p - S_q) * C_2) \bmod p) * q$. Figure 2 shows the CRT exponentiation flow as used in the current OpenSSL version (OpenSSL-0.9.8).

Modular exponentiation is based on a sequence of modular multiplications. These are typically, and in particular in OpenSSL, performed by using the Montgomery Multiplication (MMUL) algorithm.

input: ciphertext M, RSA key N, p, q, d, d_p, d_q, C_2
output: decrypted plaintext $A = M^d \bmod N$

procedure RSA_Mod_Exp():

// exponentiation modulo q
(1) (null, M_q) = Division(M, q) // $M_q = M \bmod q$
(2) S_q = Mod_Exp(M_q, d_q, q) // $S_q = M_q^{d_q} \bmod q$

// exponentiation modulo p
(3) (null, M_p) = Division(M, p) // $M_p = M \bmod p$
(4) S_p = Mod_Exp(M_p, d_p, p) // $S_p = M_p^{d_p} \bmod p$

// combine S_p and S_q using Garner's method:
// compute $S = S_q + (((S_p - S_q) * C_2) \bmod p) * q$
(5) $S = S_p - S_q$
(6) if $S < 0$ then $S = S + p$
(7) S = Multiplication(S, C_2)
(8) (null, S) = Division(S, p)
(9) S = Multiplication(S, q)
(10) $S = S + M_q$
(11) return $A = S$

Fig. 2. Modular RSA exponentiation procedure with CRT

2.3 Fixed Window Exponentiation

OpenSSL-0.9.8 uses the so called Fixed Window Exponentiation (FWE) algorithm with a window size of $w = 5$. To compute $M^{d_p} \bmod p$ (analogously hereafter also for q), the algorithm first computes a table of the $2^w - 1$ vectors, $V_i = M^i \bmod p$, for $1 \leq i \leq 2^w - 1$. The exponent d is then scanned in groups of w bits to read the value of the corresponding window i, $1 \leq i \leq 2^w - 1$. For each window, it performs w consecutive modular squarings followed by one modular multiplication with V_i, c.f. Figure 3.

2.4 Base Blinding

To thwart statistical side-channel attacks (e.g., [9]), OpenSSL-0.9.8 also applies a base blinding technique, as illustrated in Figure 4. A pair (X, Y) is generated, where X is a random number and $Y = X^{-e} \bmod N$. Then, instead of directly computing $M^d \bmod N$, the base M is first modular-multiplied by Y and the exponentiation computes $(M*Y)^d \bmod N$ (e.g., by using the CRT). After the exponentiation, the result is modular-multiplied by X to obtain the desired result, because $X*(M*Y)^d \bmod N = M^d \bmod N$. Note that if (X, Y) is a proper pair, then $(X^2 \bmod N, Y^2 \bmod N)$ is also a proper pair. Therefore, once a proper blinding pair is generated,

input: M, d, N ($M < N$, N is odd, and d is $k * w$ bits long)
output: $M^d \bmod N$

notation: $d = d[k * w - 1] \ldots d[0]$, where $d[0]$ is the least significant bit of d

Procedure:
// computation of the table values
$V_1 = M$
for i **from** 2 **to** $2^w - 1$
    $V_i = V_{i-1} * M \mod N$

// actual exponentiation phase
$S = 1$
for i **from** $k - 1$ **to** 0 **do**
    $S = S^{2^w} \ (mod\ N)$
    // scanning the window value
    $wvalue = 0$
    for j **from** $w - 1$ **to** 0 **do**
        $wvalue = wvalue * 2$
        if $d[i * w + j] = 1$ **then** $wvalue = wvalue + 1$
    // multiplication with the table entry
    $S = S * V_{wvalue} \mod N$
return S

Fig. 3. Fixed Window Exponentiation Algorithm

subsequent new pairs can be obtained for subsequent message decryptions, by performing only two modular squaring operations. By default, OpenSSL refreshes the blinding pair every 32 exponentiations.

The multiprecision operations of OpenSSL are handled by a multiprecision arithmetic library called BIGNUM. The implementations in BIGNUM library also introduces SBPA vulnerabilities to OpenSSL. The complete details of BIGNUM library are out of the scope in this paper. Yet, we outline the SBPA vulnerabilities introduced by this library after presenting a new attack on the BEEA in the next section.

3 The Main Result: Modular Inversion Via Binary Extended Euclidean Algorithm Succumbs to SBPA

Modular inversion operation is at the heart of public key cryptography. The most frequently used algorithm for modular inversion is the well known Extended Euclidean Algorithm (EEA). Due to the "unpleasant" division operations which are heavily used in the EEA, it is often substituted by another variant called the Binary Extended Euclidean Algorithm (BEEA), cf. [12]. BEEA replaces the complicated divisions of the EEA by simple right shift operations. It achieves performance advantages over the classical EEA and is especially efficient for large key-lengths (around 1024-2048 bits). The BEEA is indeed used by OpenSSL for these bitlengths.

The performance advantage of BEEA over the classical EEA is obtained via a "bit-wise-scanning" of its input parameters. As we show here, in the presence of SBPA, this property opens a new side-channel that allows an easy reconstruction of unknown input parameters to the BEEA.

To derive our new SBPA-based side-channel attack against the BEEA, we start from the classical BEEA as described in [12] and illustrated in Figure 5.

Input: c (ciphertext), rsa (RSA parameters: N, p, q, d, d_p, d_q, C_2)
Output: m (decrypted plaintext)

Procedure:
(1) (X, Y) = Get_Blinding(rsa) // X is a random; $Y = X^{-e}\ mod\ N$
(2) c = Blinding_Convert(X, c) // $c = Y * c\ mod\ N$
(3) m = RSA_Mod_Exp(c, rsa) // $m = c^d\ mod\ N$
(4) m = Blinding_Invert(Y, m) // $m = X * m\ mod\ N$
(5) padding m and output m

Fig. 4. RSA decryption procedure with base blinding, as implemented in OpenSSL

The correctness of the BEAA (and also of the EEA) relies on the fact that the following equations

$$x \cdot A + y \cdot B = u \tag{1}$$
$$x \cdot C + y \cdot D = v \tag{2}$$

hold at any iteration step of the algorithm. In the case where $\gcd(x, y) = 1$, we also have at the termination

$$x \cdot A \equiv 0 \bmod y \tag{3}$$
$$x \cdot a \equiv 1 \bmod y, \tag{4}$$

which means that a is the inverse to x modulo y.

We note here that the algorithm may terminate with a $C < 0$, which would need in practice one additional step, namely $a := x + C$, to assure that $a \in \{0, \ldots, y - 1\}$. However, since this conditional final end addition is of no further interest because as it does not affect our unknown input reconstruction problem, we ignore it for the rest of our discussion. Another important detail to point out is that for computing modular inversion, i.e., $\gcd(x, y) = 1$, the **while**-loop in Step 2 is never executed.

To derive our SBPA-based side-channel attack, we extract from the BEEA flow only the serial internal information flow regarding the branches that depend on the input values u, v, and their difference $u - v =: z$. This flow is visualized in Figure 6, where we simply "serialized" all the steps that are relevant to our case.

¿From Figure 6, we identify that the 4 critical input-dependent branches that produce 4 important information leaks. These can be put into two groups:

1. Number of right-shift operations performed on u.
2. Number of right-shift operations performed on v.
3. Number of subtractions $u := u - v$.
4. Number of subtractions $v := v - u$.

Thus, assuming that the branches in the BEEA flow are known due to a spy employing SBPA, the backward reconstruction of u and v from the termination values $u = 0$ and $v = 1$ is possible, as in Figure 3.

To come up with a concrete reconstruction algorithm, we confine the 4 branches 1, 2, 3, and 4 into successive information leakage groups. A single iteration group comprises of the information whether branch 3 or branch 4 was executed, and additionally the information on how often branch 1 and branch 2 was executed in this group. This constitutes one group — the next group starts again with entering branch 3 or branch 4. That is, for $i = 1, \ldots, \ell$ and $j = 1, 2$, we define

$$\mathcal{SHIFTS}_j[i] := \#\{\text{group } i \text{ iterations spend in branch } j\},$$

input: integers x, y
output: integers a, b, v such that $ax + by = v$, where $v = \gcd(x, y)$

1. $g := 1$;
2. **while** x and y are even **do**
    $x := x/2$, $y := y/2$, $g := 2g$;
3. $u := x$, $v := y$, $A := 1$, $B := 0$, $C := 0$, $D := 1$;
4. **while** u is even **do**
    $u := u/2$;
    if $A \equiv B \equiv 0 \bmod 2$ **then**
        $A := A/2$, $B := B/2$
    else
        $A := (A + y)/2$, $B := (B - x)/2$;
5. **while** v is even **do**
    $v := v/2$;
    if $C \equiv D \equiv 0 \bmod 2$ **then**
        $C := C/2$, $D := D/2$
    else
        $C := (C + y)/2$, $D := (D - x)/2$;
6. **if** $u \geq v$ **then**
    $u := u - v$, $A := A - C$, $B := B - D$
  else
    $v := v - u$, $C := C - A$, $D := D - B$;
7. **if** $u = 0$ **then**
    $a := C$, $b := D$, and **return**$(a, b, g \cdot v)$
  else
    goto 4;

Fig. 5. Binary Extended Euclidean Algorithm

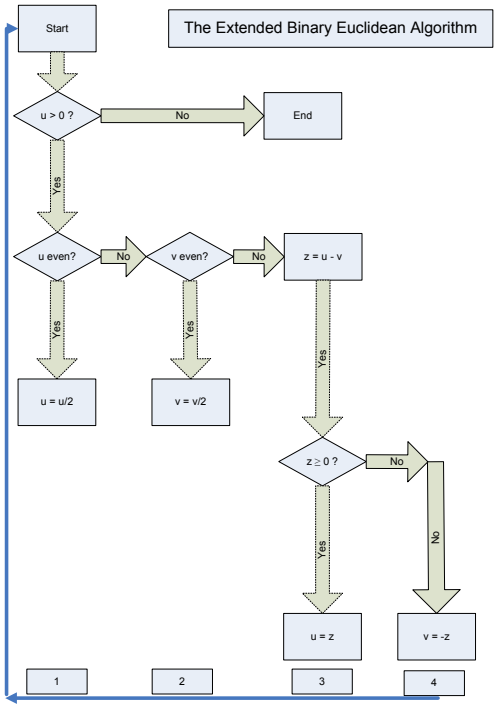

Fig. 6. The information flow of the BEAA regarding u and v.

Branch	Reconstruction operation
1	$u := 2 \cdot u$
2	$v := 2 \cdot v$
3	$u := u + v$
4	$v := v + u$

and

$$\mathcal{SUBS}[i] := \begin{cases} \text{“}u\text{”} : \text{if branch 3 is taken} \\ \text{“}v\text{”} : \text{if branch 4 is taken} \end{cases}$$

Armed we the leaked information $\mathcal{SHIFTS}_j[\cdot]$ and $\mathcal{SUBS}[\cdot]$ which are obtained by a (perfect) SBPA, our task is now to reconstruct x, y and W, X, Y, Z from the known final values $u_\mathrm{f} = 0$ and $v_\mathrm{f} = 1$ such that

$$\begin{aligned} u_\mathrm{f} \cdot W + v_\mathrm{f} \cdot X &= x \\ u_\mathrm{f} \cdot Y \ + v_\mathrm{f} \cdot Z &= y, \end{aligned}$$

Here, for the sake of clarity, we explicitly keep the value u_f and W and Y, although they actually vanish due to $u_\mathrm{f} = 0$. The corresponding algorithm, illustrated in Figure 7, accomplishes the task.

Thus, by the above exposition we have given a proof of the main contribution of the present paper.

Theorem 1. *For unknown numbers x and y with $\gcd(x, y) = 1$ as inputs to the BEEA, the side-channel information $\mathcal{SHIFTS}_j[\cdot]$ and $\mathcal{SUBS}[\cdot]$ (which can be obtained by a perfect SBPA) can be used to completely reconstruct, in polynomial time, both x and y.*

In the following sections, we show how this SBPA-enabled reconstruction theorem can be used to compromise the security of the OpenSSL RSA-CRT version in several points of the algorithm, and moreover — even in the presence of CA side-channel mitigations.

input: arrays $\mathcal{SHIFTS}[\cdot]$ and $\mathcal{SUBS}[\cdot]$ and ℓ
output: integers W, X, Y, Z as above

1. $W := 1, X := 0, Y := 1, Z := 0$;
2. $i := \ell$;
3. for $i = \ell$ **down to** 2 **do**
  if $\mathcal{SUBS}[i] =$ "u" **then**
    $W := W << \mathcal{SHIFTS}_1[i], X := X << \mathcal{SHIFTS}_2[i],$
    $W := W + Y, X := X + Z$
  else
    $Y := Y << \mathcal{SHIFTS}_1[i], Z := Z << \mathcal{SHIFTS}_2[i],$
    $Y := W + Y, Z := X + Z;$
4. $Y := Y << \mathcal{SHIFTS}_1[1], Z := Z << \mathcal{SHIFTS}_2[1];$
5. return(W, X, Y, Z)**;**

Fig. 7. BEEA reconstruction from SBPA information

4 New SBPA Vulnerabilities in OpenSSL

4.1 On the Granularity of SBPA And The Actual Threat of SBPA For Almost All Platforms

To illustrate the feasibility of attacking the BEEA algorithm with an SBPA spy or other code constructions inside openSSL, we provide here some details on how, and at what granularity, a spy process (or processes) can actually exploit the environment and extract information.

Acıiçmez et al. showed that a carefully written spy process running simultaneously with an RSA process, is able to collect almost all of the secret key bits during a *single* RSA decryption execution. They demonstrated the attack on a simplified S&M RSA implementation, which means that the branches in that demonstration were separated by thousands of cycles from each other. Moreover, this attack ran a spy that relied on the simultaneous-multithreading (SMT) capability of the tested platform.

However, we point out here that the actual power of SBPA is most probably not limited to this basic application where branches are widely separated, and not limited to SMT capable platforms. In general, SBPA has the potential to reveal the entire execution flow of a target process in *almost any* execution environment — with or without SMT. We now explain the grounds for this claim.

MicroArchitectural attacks exploit shared microprocessor components to compromise security systems by using spy processes. They require a quasi-parallel execution of the spy and the "crypto" processes on the same processor. Although a hardware-assisted SMT feature seems to be mandatory to achieve this, recent studies indicate the opposite [13, 14]. Neve et al. showed that it is possible to exploit some of the scheduling functionality of operating system (OS) to accomplish the same effect on a non-SMT platform — that is, as if the spy and crypto processes ran simultaneously on an SMT platform. In fact, the idea of exploiting OS scheduling for side-channel attacks by using a spy process and a trojan process, was published already in 1992 [10].

Their method of transferring SMT-based MicroArchitectural attacks to non-SMT systems rely on the preemptive OS scheduling property, which is the way that on OS handles multitasking capability. Multitasking operating systems allow for the execution of multiple processes on the same computer, concurrently. In other words, each process is given permission to use the processor's resources, including the cache, branch prediction unit, and other MicroArchitectural resources. When several different processes run actively on the system, the OS assigns a certain maximum execution time (called quantum) to the process that gets his turn to execute.

The preemptive OS scheduling property gives a process the ability to trigger a context switch, i.e., yielding the CPU to another process, at any desired time before consuming the entire quantum. The interesting and security-critical point is what the OS does afterwards. It indeed schedules the next process in the queue, but lets it run only for the duration of the remaining part of the quantum that was mostly consumed by the prior process. Neve et. al. used this OS functionality to stretch the very short execution time of the AES over several quantums and could therefore obtain spy measurements precisely centered around a very small number of instructions.

Using this result, we can safely speculate that it would be possible to apply SBPA attacks at a fine-granularity of even a few hundred cycles and — in an idealized scenario — to detect *all* the branches of an attacked application. This dramatically broadens the scope of SBPA attacks, for example to attack the BEEA procedure.

We illustrate a possible attack path. A spy process can run until the end of a quantum, but just before the end of this quantum — e.g., a couple of hundred cycles — trigger a context switch. Then, the OS would let the crypto process run for a very short time — the remaining time till the end of that quantum — and give the execution back to the spy. In fact, an attacker can even use another "trojan process" to handle switching spying independently.

This method still does not allow us to analyze branches at the granularity of a few cycles. However, note that the arithmetical and logical operations in RSA are multi-precision, meaning that the operands are several machine-words long. Any such computation in RSA requires execution of tens to hundreds of instructions. Furthermore, the actual implementation of the BBEA algorithm in OpenSSL involves function calls, which inherently consume overhead cycles. At the minimum, these functions declare and initiate some variables, and need to execute some steps (in a loop). All these steps together most probably will leave a sufficiently large window for a successful SBPA attack.

Therefore — although we have not developed code to demonstrate this attack — we strongly conjecture that an SBPA attack, coupled with this "OS trick", has the potential of revealing the entire execution flow of the BEEA on almost any platform. As we show in the following section, if such an "ideal SBPA attack" is possible, and if a spy process successfully enters the attacked system, this can compromise the security of the RSA implementation of OpenSSL at several points.

4.2 New SBPA Vulnerabilities in OpenSSL Due to Theorem 1

In this section we describe several vulnerabilities that we have identified in openSSL, in the context of SBPA attacks and Theorem 1 that shows how the inputs to the modular inversion procedure can be extracted by an SBPA spy. Referring back to Section 2, we can see that modular inverses are computed during the RSA computations when:

1. the private exponent $d = e^{-1} \bmod (p-1)(q-1)$ is computed,
2. the CRT parameter $q^{-1} \bmod p$ is computed,
3. the Montgomery constant $-p^{-1} \bmod m$ (e.g. with $m = 2^{32}$) is computed, and
4. the masking pair (X, $Y = X^{-e} \bmod N$) is computed.

Now, the implications of Theorem 1 become clear. In cases 1-3, an SBPA attack can directly compromise the secret RSA keys. In case 4, an SBPA attack can compromise the blinding mitigation. If the blinding factor is revealed, the blinding mitigation technique to thwart statistical attacks collapses. In such a scenario, other and simpler attacks could be launched. If the SBPA spy indeed compromises the blinding mitigation, the situation is effectively a chosen-plaintext scenario, where the timing depends on the chosen-plaintext and the secret primes. Clearly, a remote attack can now be mounted on the application. We mention here two further attack scenarios: First, remote timing attacks [9] that exploit timing differences due to the End Reduction (ER) step in the Montgomery multiplications, and second remote timing attacks that exploit the early exit short cut taken in the division algorithm. We add some details to explain the second attack in the following.

Remote timing attacks that exploit the early exit short cut taken in the division algorithm. The first step in the CRT algorithm, is reducing the message M modulo p and q. This reduction is carried out by the BIGNUM Division function. There are two branches in the Division procedure of OpenSSL-0.9.8, that leak information that could potentially compromise the private key. Given A and B, Division (A, B) performs the following:

a. If $A < B$, Division (A, B) immediately returns the quotient 0, remainder B, and exits.
b. If the divisor B has t words, and the first t words of A form a number which exceeds B, the following is done, where b is the bitlength of B: $2^{tw} < A < 2^{b+tw}$, and if it is satisfies, Division (A, B) resorts to some shortcut path.

It follows that the execution time of Division is data dependent and could be used very simple in a binary-search oriented algorithm to find p or q.

4.3 Vulnerability Due to Exponent Scanning in The FWE Procedure

The FWE algorithm is in theory immune to SBPA. Unlike S&M or SWE algorithms, it does not involve exponent-dependent execution because the modular multiplications/squares are independent of the exponent value. However, the actual code implementation of FWE in OpenSSL-0.9.8 makes it potentially susceptible to SPBA attack. The routine that constructs the "window value", while scanning the exponent string does it via a bit-by-bit, i.e., an if-then-else method. The code invokes the function BN_is_bit_set(a, n) whose task is to read the value of the n^{th} bit of a. Actually, BN_is_bit_set is currently implemented as follows:

```
int BN_is_bit_set(const BIGNUM *a, int n)
{
   int i,j;

   bn_check_top(a);
   if (n < 0) return 0;
   i=n/BN_BITS2;
   j=n%BN_BITS2;
   if (a->top <= i) return 0;
   return((a->d[i]&(((BN_ULONG)1)<<j))?1:0);
}
```

If the conditional statement in the last line of this function is actually performed (i.e., not removed by the compiler), it can be exploited by an SBPA spy to completely discover the secret exponent that is being scanned.

5 Software Mitigations to Protect OpenSSL Against The SBPA Vulnerabilities

Our general methodology for creating an SBPA-aware RSA implementation is to first assume that SBPA can reveal the entire flow of the execution. Under this assumption, we scan through all steps of the algorithm to detect the conditional branches. For each branch we consider the following question: *if an attacker obtains the complete history of this branch, would this provide him with useful information?* Whenever the result to this question is positive, we consider fixing the problem by either changing the algorithm flow to eliminate the conditional execution, or by assuming that this branch is executed in an off-line environment (i.e., before any spy can enter the system).

5.1 Fixing the Exponent Scanning Vulnerability

Mitigating this vulnerability is simple: the conditional statement in the last line BN_is_bit_set(a, n) needs to be removed. Since BN_is_bit_set(a, n) checks whether the input bit is 0 or 1, the function can simply return the value of that bit. We therefore propose the following change in the code:

```
Current Version:  return((a->d[i]&(((BN_ULONG)1)<<j))?1:0);
Our Proposal:     return((a->d[i] >> j) & 0x01);
```

Remarks:

1. A more efficient way to handle the exponent scanning is to avoid the call to BN_is_bit_set function. The window values can be constructed by directly copying the corresponding section of the exponent string at once (using shifts and logical operations) instead of doing it in a bit-by-bit fashion.

2. The secret exponents d_p and d_q are fixed for a given RSA key. Thus, the window values can be computed once, during the key generation phase, stored, and then used per each exponentiation without re-scanning the exponent string.
3. A good compiler, particularly in an optimized mode, should be able to identify and to automatically eliminate the discussed branch. In fact, when OpenSSL-0.9.8 is compiled with a gcc compiler (using Linux operating system), the compiler indeed removes the branches. However, this compiler optimizing property cannot be guaranteed for any compiler in any mode, and should not be considered as an automatic mitigation.

5.2 A Minimal Set of Mitigation for CRT-RSA and Key Generation

In this section, we outline a few pin-pointed changes to OpenSSL-0.9.8, that can be used for achieving (what seems to be) an SBPA-protected implementation.

The proposed changes are in the Big Numbers library of OpenSSL, particularly in the functions BN_mod_inverse(in, a, n) and BN_div (dv, rm, num, $divisor$).

1. **Avoiding the use of BEEA for modular inversion in BN_mod_inverse.** For odd n, whose bit-length is less than 450 (for a 32-bit machine) or less than 2048 bit (for a 64-bit machine), the function BN_mod_inverse(in, a, n) uses the BEEA. In other cases, it uses a general inversion algorithm that uses the division as implemented in BN_div(dv, rm, num, $divisor$). An easy mitigation technique is to eliminate the branch that allows for selecting BEEA, thus to always use the general inversion via Division.
2. **Avoiding other branches during modular inversion that uses BN_div**. To optimize performance the general inversion algorithm first checks if the divisor and dividend are close and in that case does not invoke an BN_Division (performing the division in another method). Further, in intermediate steps that require multiplication, the function first checks if the multiplier is between 1 and 4 and in that case performs additions instead of invoking BN_mul. These shortcuts can be eliminated, in order to avoid leaking any information associated with the relative sizes of the operands (which could potentially facilitate an attack). The early exit step in the division algorithm should also be eliminated. We point out that these branch elimination steps can be spared if the base blinding mitigations are assumed to successfully mask the side-channel information.

We emphasize that in general, a minimal quick-fix approach for handling the security of a cryptographic implementation is a less recommended approach than a fundamental revision of the CRT-RSA procedures which should be seriously considered. We propose such an approach in the paper's next part.

5.3 Intrinsically BPA-Protected CRT-RSA Implementation — Smooth CRT-RSA

Our goal here is to outline a CRT-RSA implementation that does not contain conditional branches. To this end, we need to eliminate modular reductions, divisions, and the conditional ER step from the computation of RSA at all. Our proposed approach, which we call "Smooth CRT-RSA" achieves this objective, and is therefore an intrinsically SBPA-protected CRT-RSA implementation.

There are several methods one can use to eliminate the conditional ER step (e.g., [11], [18]). The reason why we need to perform the ER step at the end of a Montgomery multiplication is that the result of the algorithm returns a result that is smaller than twice the modulus, but not necessarily smaller than the modulus itself. Thus, the result sometimes needs to be reduced by means of one subtraction of the modulus (hence called an ER step). However, if the data structures used in an CRT-RSA implementation can accommodate the possibility of storing values between p and $2p$ (q and $2q$, resp.), then the ER step can be simply avoided. There are two ways to achieve this:

1. Increasing the size of the data structures by one word.
2. Decreasing the size of p and q by two bits.

Note that if the ER step is eliminated, and if the procedure that implements MMUL is written in a way that does not introduce superfluous branches, then the base blinding mitigation is not necessary.

To eliminate the need for modular reductions and divisions in CRT-RSA computation, we propose to change the CRT-RSA flow, and introduce three new variables H_{3p}, H_{3q}, and MC_2:

$$H_{3p} = 2^{3k} \bmod p, \quad H_{3q} = 2^{3k} \bmod q, \quad MC_2 = q^{-1} * 2^k \bmod p \tag{5}$$

where k is defined by $2k = n$.

These parameters can be easily derived from the traditional Montgomery constants $H_p = 2^{2k} \bmod p$, $H_q = 2^{2k} \bmod q$, and from the CRT parameter $C_2 = q^{-1} \bmod p$ which are currently used in OpenSSL for converting the exponentiation base from the integer to the Montgomery domain. Only one Montgomery multiplication is required for this derivation:

$$\begin{aligned} H_{3p} = 2^{3k} \bmod p &= MMUL(H_p, H_p, p) \\ H_{3q} = 2^{3k} \bmod q &= MMUL(H_q, H_q, q) \\ MC_2 = q^{-1} * 2^k \bmod p &= MMUL(C_2, H_p, p) \end{aligned}$$

These computations can be included in the Montgomery initialization phase, and the new parameters can replace the old ones. Figure 8 illustrates the new proposed smooth CRT-RSA flow. Interestingly, the Smooth CRT-RSA has a better performance than the standard CRT-RSA that is implemented in OpenSSL-0.9.8.

Input: M (ciphertext)
Input: N (modulus; n=2k bits long)
Input: p (secret prime; k bits long)
Input: q (secret prime; k bits long)
Input: d_p (secret CRT exponent for the modulus p)
Input: d_q (secret CRT exponent for the modulus q)
Input: MC_2 (= $q^{-1} * 2^k \ mod \ p$)
Output: $A = M^d \ mod \ N$ // d is the RSA private exponent

Pre-computed constants:
$H_{3p} = 2^{3k} \ mod \ p$
$H_{3q} = 2^{3k} \ mod \ q$
// Procedure: $MontExp(A, X, N) = A^X 2^{-n(X-1)} \pmod N$
// (modular exponentiation steps with modular multiplications replaced by $MMUL$ operations)
// Procedure: $MontReduce$ — Montgomery reduction.

Setup:
Conversion into Montgomery Domain:
(1) R_1 =MontReduce (M, p) // $R_1 = M * 2^{-k} \ mod \ p$
(2) R_1 =MMUL (R_1, H_{3p}, p) // $R_1 = M * 2^k \ mod \ p$
(4) R_2 =MontReduce (M, q) // $R_2 = M * 2^{-k} \ mod \ q$
(5) R_2 =MMUL (R_2, H_{3q}, q) // $R_2 = M * 2^k \ mod \ q$

Exponentiations:
(3) M_1 =MontExp (R_1, d_p, p) // $M_1 = R_1^{d_p} 2^{-k(d_p-1)} \ mod \ p = R_1^{d_p} 2^k \ mod \ p$
(6) M_2 =MontExp (R_2, d_q, q) // $M_2 = R_2^{d_q} 2^{-k(d_q-1)} \ mod \ q = R_2^{d_q} 2^k \ mod \ q$
Conversion from Montgomery Domain:
(5) M_1 =MMUL (M_1, 1, p) // $M_1 = M^d \ mod \ p$
(5) M_2 =MMUL (M_2, 1, q) // $M_2 = M^d \ mod \ q$

Recombination with MMUL:
(7) M_2 = M_2 - M_1 //
(8) if($M_2 < 0$) M_2 = M_2 + p //
(9) R_1 =MMUL(M_2, MC_2, p) //
(10) $R_1 = M_2 * p$ //
(11) $A = R_1 + M_1$ //
(12) return A //

Fig. 8. Smooth CRT-RSA

6 Conclusions

Our main result in this paper was the unexpected novel SBPA side-channel attack against the public-key cryptography primitive BEEA that is used for modular inversion. This started a research path for fully exploring the potential risk of SBPA attacks.

Since deactivating branch prediction units on all general purpose platforms, or disabling multiprocessing capabilities in the OS, is clearly an unattractive mitigation approach, one conclusion is that cryptographic software that run on general platforms needs to be (re-)written in an SBPA-aware style. Thus, another focus of our paper was on a software mitigation methodology for protecting applications against SBPA attacks. In

order to do so, we also highlighted — on top of the central SBPA-enabled BEEA side-channel attack — some other very obvious OpenSSL code constructs which are susceptible to SBPA attacks. We then proposed several simple techniques to mitigate these vulnerabilities, which could be applied to future versions of OpenSSL towards an SBPA-aware version.

The minimal set of mitigations that were detailed in Section 5.2 have already been implemented by Intel Corporation's security experts, and will be contributed to the open source community shortly, in order to facilitate their quick deployment into a coming version of OpenSSL. These modifications carry only a small performance penalty.

In addition, we have also im implemented the Smooth RSA proposal from Section 5.3 on top of the minimal set. As expected, Smooth RSA and the minimal set of mitigations not only protects OpenSSL 0.9.8a from the SBPA attacks, but also *improves* the performance due to the elimination of some operations such as the ER step in the Montgomery multiplications. The details, including the comparative performance results, will soon be available in a subsequent paper.

We also point out that the present paper does not claim an exhaustive SBPA side-channel security analysis with a full proof on corresponding software countermeasures for OpenSSL. It rather indicates that this task is required, especially since SBPA attacks potentially threatens every step of the OpenSSL code. However, the present paper made a major step towards this goal.

Another conclusion of the paper is that the PC-oriented side-channel attack field is very subtle due to the high MicroArchitectural complexity of today's processors. Thus, we expect that new MicroArchitectural attacks will inevitably be discovered in the future.

Last but not least, we summarize the current situation as follows. Probably the best defense against current and future MicroArchitectural attacks is to let the cryptographic software community become aware of the security implications of writing cryptographic software that is going to be executed on throughput-optimized general purpose processors. This will help software be written using a proper side-channel attack aware methodology.

Acknowledgements

The authors are members of the Applied Security Research Group at The Center for Computational Mathematics and Scientific Computation within the Faculty of Science and Science Education at the University of Haifa (Israel).

References

1. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting Secret Keys via Branch Prediction. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pages 225-242, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.
2. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The Power of Simple Branch Prediction Analysis. *ACM Symposium on Information, Computer and Communications Security — ASIACCS'07*, 2007, to appear.
3. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The Power of Simple Branch Prediction Analysis. Cryptology ePrint Archive, Report 2006/351, October 2006.
4. O. Acıiçmez, W. Schindler, and Ç. K. Koç. Cache Based Remote Timing Attack on the AES. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pages 271-286, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.
5. O. Acıiçmez, W. Schindler, Ç. K. Koç. Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations. *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security*, C. Meadows and P. Syverson, editors, pages 139-146, ACM Press, 2005.
6. O. Acıiçmez, J.-P. Seifert, and Ç. K. Koç. Predicting Secret Keys via Branch Prediction. Cryptology ePrint Archive, Report 2006/288, August 2006.
7. D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005. Available at: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
8. D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. *Advances in Cryptology - CRYPTO '98*, H. Krawczyk, editor, pages 1-12, Springer-Verlag, Lecture Notes in Computer Science series 1462, 1998.

9. D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Proceedings of the* 12^{th} *Usenix Security Symposium*, pages 1-14, 2003.
10. W. M. Hu. Lattice scheduling and covert channels. *Proceedings of IEEE Symposium on Security and Privacy*, IEEE Press, pages 52-61, 1992.
11. S. Gueron. Enhanced Montgomery Multiplication. Lecture Notes Comp. Sci. (CHES 2002) **2523**, pages 46-56, 2002.
12. A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1997.
13. M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. *Selected Areas of Cryptography — SAC'06*, to appear.
14. M. Neve. Cache-based Vulnerabilities and SPAM Analysis. Ph.D. Thesis, Applied Science, UCL, July 2006
15. 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, Lecture Notes in Computer Science series 3860, 2006
16. Openssl: the open-source toolkit for ssl / tls.
17. C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005.
Available at: `http://www.daemonology.net/hyperthreading-considered-harmful/`
18. C. D. Walter. Montgomery exponentiation needs no final subtractions. Electronics Letters **35** pages 1831 – 1832(1002)