Published in J. Hernando, E. Fernndez-Medina, and M. Malek, Eds, *International Conference on Security and Cryptography – SECRYPT 2007*, pp. 189-196, INSTICC Press, 2007.

SECURING OPENSSL AGAINST MICRO-ARCHITECTURAL ATTACKS

Marc Joye
Thomson R&D France, Technology Group, Corporate Research, Security Laboratory,
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
marc.joye@thomson.net

Michael Tunstall
Department of Electrical & Electronic Engineering, University College Cork, Cork, Ireland
miket@eleceng.ucc.ie

Keywords: RSA, Modular exponentiation, Micro-architectural attacks, Side-channel resistant implementations.

Abstract: This paper presents a version of the 2^k-ary modular exponentiation algorithm that is secure against current methods of side-channel analysis that can be applied to PCs (the so-called micro-architectural attacks). Some optimisations to the basic algorithm are also proposed to improve the efficiency of an implementation. The proposed algorithm is compared to the current implementation of OpenSSL, and it is shown that the proposed algorithm is more robust than the current implementation.

1 INTRODUCTION

Exponentiation algorithms are important for many public-key cryptographic algorithms, in particular for computing the modular exponentiation necessary for RSA (Rivest et al., 1978). It is therefore essential to ensure that implementations of algorithms requiring a modular exponentiation are not vulnerable to any known attacks.

Side-channel attacks can be applied remotely to a PC, by observing the time taken for a processor to compute a given function. In addition, they may observe some micro-architectural features, e.g. the cache or branch predictor of a processor which is executing the function. This usually requires the execution of a spy process to observe and manipulate a processor while it is running. A more detailed description of different types of side channels that can be applied to PCs is given in Section 2.

This paper proposes a modified 2^k-ary modular exponentiation algorithm (the notation used in this paper is taken from (Knuth, 2001)). The proposed algorithm is resistant to all currently known side channels available to an attacker targeting a PC implementation. The security of this algorithm is analysed in terms of its side-channel resistance to the attack methods presented in Section 2, and some further optimisations to the basic algorithm are also presented.

The proposed algorithm is compared to the current secure implementation of modular exponentiation used in OpenSSL (OpenSSL, 2007). It is demonstrated that some bits of the private exponent risk being revealed if an attacker is able to modify the cache and observe the effect on the output. The proposed algorithm is shown to be more robust than the current OpenSSL implementation.

The rest of this paper is organised as follows. The different side channels that can potentially be exploited to reveal secret information are described in Section 2. The proposed exponentiation algorithm is described in Section 3, and some further optimisations are presented in Section 4. A comparison of the proposed algorithm with the implementation used in the current version of OpenSSL is presented in Section 5. This is followed by our conclusions in Section 6.

Notation: The base of a value is determined by a trailing subscript, which is applied to the whole word preceding the subscript. For example, $\mathtt{FE}_{16}$ is 254 expressed in base 16, $d = (d_{\ell-1}, d_{\ell-2}, \ldots, d_0)_2$ gives a binary expression for d, and $d = (d_{\ell-1}, d_{\ell-2}, \ldots, d_0)_4$ gives an expression where each d_i, for $0 \leq i < \ell$, represents two bits of d.

In all the algorithms described in this paper λ represents the Carmichael function, where $\lambda(N)$ is defined for N as the smallest positive integer m such that $a^m \equiv 1 \pmod{N}$ for every integer a that is co-

prime to N. In particular, if $N = pq$ is an RSA modulus then $\lambda(N) = \text{lcm}(p-1, q-1)$. The notation ϕ represents Euler's totient function, where $\phi(N)$ equals the number of positive integers less than N which are coprime to N. If $N = pq$ is an RSA modulus then $\phi(N) = (p-1)(q-1)$.

2 SIDE-CHANNEL ANALYSIS

There are several different methods of side-channel analysis that can potentially be applied to an implementation of the RSA signature scheme. These methods are summarised below.

2.1 Timing Analysis

The first academic publication of side-channel analysis was an attack that observed the correlation between guessed bits of a secret and the time required to compute an algorithm (Kocher, 1996). The principle target of the timing analysis described was the RSA signature scheme. This was extended in (Schindler, 2000) to include the RSA signature scheme when it is calculated using the Chinese Remainder Theorem (Knuth, 2001) and Montgomery multiplication (Montgomery, 1985). These attacks were typically thought of in terms of smart cards, where it is trivial to observe the execution time of a naïvely implemented process.

It was demonstrated in (Brumley and Boneh, 2003) that timing analysis of the computation of RSA signatures could be conducted across a network against complex implementations, such as OpenSSL. This demonstrated the need to consider the possible side channels that could be exploited in implementations of cryptographic algorithms on all platforms.

In this paper it is assumed that the underlying multiplication algorithm used in the exponentiation algorithm is resistant to timing analysis. For example, if we consider Montgomery multiplication, which contains a conditional modular subtraction, it is pointed out in (Hachez and Quisquater, 2000; Walter, 1999a; Walter, 1999b) that this final operation can be omitted.

2.2 Cache-Based Side-Channel Analysis

A cache is a small, fast RAM memory whose role is to buffer the lines of Non-Volatile Memory (NVM) or external RAM being fetched. When a data or instruction word is to be fetched from the NVM or external RAM, the CPU will first check whether this particular word is already in the cache: if yes (this is a *cache hit*), the word is fetched directly from the cache. If, on the contrary, this particular word is not cached this is a *cache miss*. The CPU will then fetch a whole line (e.g. 32 bytes) within which the targeted word is found. The data in this cache line can then be accessed rapidly by the CPU, whereas accessing external resources to fetch data takes significantly more time.

Using the cache as a side channel to attack an implementation of a cryptographic algorithm was first proposed in (Tsunoo et al., 2003). Several attacks have since been published using cache access events as a side channel (Bernstein, 2005; Bertoni et al., 2005; Osvik et al., 2006) to derive a secret key used in implementations of block ciphers, such as DES and AES. These examples are predominately a specific case of timing analysis, where the total number of cache misses in an algorithm is used to determine information on the secret key being used.

Another example of using the cache as a side channel has been termed trace-driven cache analysis, and was first described in (Page, 2002). This attack functions by observing what cache lines are used by a process computing a cryptographic algorithm. This is possible as the cache is open to inspection and modification by all processes being run on a PC. Implementations of attacks that exploit this method of side-channel analysis against PC implementations of AES are described in (Acıiçmez and Koç, 2006; Osvik et al., 2006).

2.3 Branch Prediction Analysis

Modern chips for PCs include branch prediction to improve overall performance. This involves the inclusion of a Branch Target Buffer (BTB) and a Branch Predictor (BP). The BTB is a buffer of limited size that acts as a cache for storing the addresses of previously executed branches. The BP is an algorithm that attempts to predict what branches will be taken, based on previous observations. If a conditional branch is present in an algorithm (e.g. an **if** command) the BP will attempt to predict the outcome of this branch and load the relevant instructions into the CPU. If the prediction is correct this increases performance, since the relevant instructions are available. However, if the prediction is incorrect the CPU is obliged to fetch the instructions for the other branch. In (Acıiçmez et al., 2007c) it is pointed out that this will lead to a difference in execution time and can therefore be used to conduct a timing analysis.

More sophisticated attacks are presented in (Acıiçmez et al., 2007b; Acıiçmez et al., 2007c) that modify the BTB to produce effects that can leak

information more efficiently than observing the time taken to compute an algorithm. Indeed, the most efficient attack described involves closely observing the BP during the computation of an RSA signature by using a spy process that modifies the BTB and observes the subsequent behaviour. This could allow an attacker to derive the private key from one signature generation. An implementation of this type of attack on a modified version of the function used in OpenSSL to generate RSA signatures is described in (Acıiçmez et al., 2007b).

Again, it is assumed that the underlying multiplication algorithm is not vulnerable to this type of side-channel analysis, i.e. there are no conditional branches in the multiplication algorithm and each multiplication involves exactly the same number of operations for inputs of a given bit length.

3 SIDE-CHANNEL RESISTANT 2^k-ARY EXPONENTIATION

The algorithm proposed in this paper is a modified 2^k-ary exponentiation, as defined in (Knuth, 2001). This is combined with the techniques used to protect embedded implementations from Differential Power Analysis (Kocher et al., 1999), where the input values are multiplied by small random values to mask the behaviour of the algorithm during execution. This algorithm is described in Algorithm 1, where ρ is a small integer that is used to increase the bit length of N so that it is the same as M^*.

The input Λ is either $\lambda(N)$ or some multiple thereof. In the case of RSA we can use $\phi(N) = (p-1)(q-1)$, or even $(e \cdot d - 1)$ (where e is the public exponent), which is a multiple of $\lambda(N)$. Note that working with $(e \cdot d - 1)$ instead of $\lambda(N)$ does not have a large impact on the performance of the algorithm, since e is usually small (typically e will be equal to 3 or $2^{16}+1$).

The variable $R[1]$ is set to a value equivalent to 1 mod N and will therefore have no effect on the result but will involve a multiplication with an integer modulo $N' = \rho \cdot N$. This means that there is no conditional branching within the exponentiation loop. The multiplication with a given $R[i]$ can be determined by calculating a pointer to the relevant variable, assuming that the variables of $R[i]$, for $1 \leq i \leq b$, are contiguous in memory.

Algorithm 1 also slightly differs from the classical 2^k-ary exponentiation algorithm as the first operation of the **while** loop is

$$A \leftarrow A^{2^{k-1}} \bmod R[0],$$

Algorithm 1: Secure 2^k-ary exponentiation algorithm

Input: M, $d = (d_{\ell-1}, d_{\ell-2}, \ldots, d_0)_b$ where $b = 2^k$ for some $k \geq 1$, N, ρ, Λ, and two random values r_1 and r_2 (of bit length $|\rho|_2$).
Output: $S = M^d \bmod N$.

$M^* = M + r_1 \cdot N$
$d^* = (d - 1 + r_2 \cdot \Lambda)/2$
$U^* = 1 + r_1 \cdot N$
$N' = \rho \cdot N$

$R[0] \leftarrow N'$
$R[1] \leftarrow U^* \bmod R[0]$
$R[2] \leftarrow M^* \bmod R[0]$

for $j = 3$ **to** b **do**
  $R[j] \leftarrow R[j-1] \cdot R[2] \bmod R[0]$
end

$i \leftarrow \lfloor \log_b d^* \rfloor$
$A \leftarrow R[d_i^*]^2 \bmod R[0]$
$i \leftarrow i - 1$
while $(i \geq 0)$ **do**
  $A \leftarrow A^{2^{k-1}} \bmod R[0]$
  $A \leftarrow A \cdot R[d_i^* + 1] \bmod R[0]$
  $A \leftarrow A^2 \bmod R[0]$
  $i \leftarrow i - 1$
end

$A \leftarrow r_1 \cdot A \cdot R[1] \bmod R[0]$
$A \leftarrow A / r_1$

return A

rather than

$$A \leftarrow A^{2^k} \bmod R[0] \ .$$

This can be explained if we suppose that

$$M_2 = M^2 \bmod N,$$

then $S = M^d \bmod N$ can be rewritten as

$$S = {M_2}^{(d-1)/2} \cdot M \bmod N \ .$$

This allows d to be replaced with (a randomised representation of) $(d-1)/2$, when it is multiplied by a small random at the beginning of the exponentiation. The last modular multiplication can be moved outside the **while** loop reducing the amount of computation required within the loop. This assumes that d is always odd, Λ is always even (as is the case for RSA), and the computation of d^* is always possible.

Each random value used has the effect that each multiplication is randomised by a value whose effect is equivalent to a multiplication by 1 mod N and is therefore easily removed at the end. The bit length of

the random values used are often determined by the algorithm and/or the architecture used. For example, in software implementations the natural choice would be to use random values with the same bit length as the words manipulated by the processor (or a multiple thereof).

The initialisation of $R[1]$ and $R[2]$ ensures that these variables always contain a value whose bit length is similar to the bit length of N'. A value with a constant bit length will, therefore, always be given to the underlying multiplication algorithm. This removes the possibility of an attacker provoking a situation that could allow timing analysis by choosing M as a small integer (chosen-message attack).

The change in d means that, for a fixed value of d, each execution of the algorithm will behave differently. It is therefore not possible to derive information by observing multiple executions, an attacker is obliged to attempt to derive d from a single execution.

Also note that the two last instructions, $A \leftarrow (r_1 \cdot A \cdot R[1] \bmod R[0])/r_1$, can also be implemented as $A \leftarrow A \cdot R[1] \bmod N$. This choice of instruction will depend on which instruction is most suitable for a given implementation.

The security of this algorithm against the side-channel analysis methods described in Section 2 is as follows.

Timing Analysis: The algorithm will take a constant number of operations to execute, i.e. $\lceil (\log_2 d^*)/k \rceil$ sets of k squaring operations and one multiplication. The only differences in computation time will be caused by the variable bit length of r_2. However, there are no data dependent differences in execution time to allow a timing analysis to take place. As described in Section 2, it is assumed that the underlying squaring operation (respectively the multiplication) will always take the same amount of time for inputs of a given bit length. The bit length of the inputs to all the multiplications is identical for all d_i^* because the initialisation steps mean that each $R[i]$, for $1 \leq i \leq b$, contains a variable with a bit length similar to N'.

Cache-Based Side-Channel Analysis: The result of the calculation of the powers of M^* will be stored in the cache. In a multi-threaded system it would be potentially possible to exploit this, by determining how an implementation behaves with different values of d. This possibility is removed by the masking of the input variable with small random variables. In particular, the modification to d means that the cache lines accessed for a given value of M will vary unpredictably from one execution to another.

If an attacker is able to produce a trace of the cache accesses it is potentially possible to determine some information on d^*, as each value of d_i^* will cause the algorithm to access different cache lines. An attacker may therefore be able to determine d^* which will give a value that is equivalent to d when used as an exponent modulo N. A trick that can remove this side channel is used in the current implementation of OpenSSL and is described in Section 5.

Branch Prediction Analysis: As mentioned previously, there is no conditional branching within the (main loop of) the algorithm, and it will, therefore, not be possible to determine any bits of d, or d^*, by observing the behaviour of the branch predictor. The required variable can be accessed by calculating an offset from the beginning of $R[0]$, if $R[0]$ to $R[b]$ are stored in contiguous memory.

It would be reasonable to assume that an attacker can determine at what point the conditional jumps used in the **for** and **while** loops occur (Acıiçmez, 2007). As stated above, it is assumed that each squaring operation (respectively the multiplication) will always take the same amount to time to calculate for inputs of a given bit length. The bit length of each $R[i]$, for $1 \leq i \leq b$, is identical, and an attacker will, therefore, not be able to derive any information by choosing M as a small integer.

This side channel can also be removed by unrolling the loops, either in the source code or by using the compiler. However, this would require the implementation of a different function for each bit length of interest, and that the most significant bit of r_2 is set to one so that the bit length of d^* is constant.

4 FURTHER OPTIMISATIONS

Another version of Algorithm 1 is presented in Algorithm 2, and contains some further optimisations that can make an implementation more efficient in terms of speed and memory required. It is possible to combine N' and $R[0]$ (as used in Algorithm 1) in memory to reduce the memory that is required to implement the proposed algorithm. This can be achieved by observing that

$$R[1] \leftarrow 1 + r_1 \cdot N \bmod N',$$

whose purpose is to allow a multiplication by 1 mod N to take place, can also be written as

$$R[1] \leftarrow r_1 \cdot N - 1 \bmod N' \quad \left(\equiv -1 \pmod N\right).$$

This is because it is always followed by a squaring (namely, $A \leftarrow A^2 \bmod R[0]$).

However, letting $N' = \rho \cdot N$, this requires that the **while** loop is modified to take into account this change in Algorithm 2. Each $R[i]$, for $1 \leq i < b$, therefore contains $M^{*i}/2 \bmod N'$ —and $R[0]$ contains a value that is equivalent to $-1/2 \pmod N$, and after the multiplication operation the result is corrected by doubling A. Provided that N is odd (which is always the case for RSA moduli), this can be implemented on a processor that manipulates words of ω bits by calculating

$$N'' = 2^{\omega-1} \cdot \rho \cdot N$$

where ρ is a small *odd* random integer that is used to increase the bit length of $N \cdot 2^{\omega-1}$ and to randomise the value of $-1/2 \pmod N$. Indeed, since N and ρ are assumed to be odd, it follows that

$$\left\lfloor \frac{N''}{2^{\omega}} \right\rfloor = \left\lfloor \frac{\rho \cdot N}{2} \right\rfloor = \frac{\rho \cdot N - 1}{2} \equiv -1/2 \pmod N$$

and $N'' \bmod 2^{\omega} = 2^{\omega-1}$. In other words, this will create a value for N'' where the least significant word is $2^{\omega-1}$ and the remaining upper words represent a randomised value for $-1/2 \pmod N$. In order to be resistant to side-channel analysis, the precomputed values of $M^{*i}/2$, for $1 \leq i < b$, are computed modulo $N' = \rho \cdot N$ and so are represented with the same number of words as $(\rho \cdot N - 1)/2$, which is written in $R[0]$.

As presented, Algorithm 2 assumes a little-endian representation; if all $R[i]$ are stored in continuous memory, $R[0]^-$ denotes the memory location starting one word before $R[0]$. Nevertheless, it can easily be adapted to accommodate a big-endian representation.

In Algorithm 2 the modulus N'' is always an even number. This excludes the use of Montgomery multiplication, and will require the use of an alternative, such as Barrett or Quisquater multiplication (Barrett, 1987; Quisquater, 1992).

Algorithm 2: Secure 2^k-ary exponentiation algorithm (II)

Input: M, $d = (d_{\ell-1}, d_{\ell-2}, \ldots, d_0)_b$ where $b = 2^k$ for some $k \geq 1$, N odd, random odd value ρ, Λ, processor word-size in bits ω, 2 random values r_1 and r_2 (of bit length $|\rho|_2$), and a random value r_3 (of bit length ω).
Output: $S = M^d \bmod N$.

$M^* = (M/2 \bmod N) + r_1 \cdot N$
$d^* = (d - 1 + r_2 \cdot \Lambda)/2$
$N' = \rho \cdot N$
$R[0] \leftarrow N'$
$R[1] \leftarrow M^*$
$A \leftarrow R[1] + R[1] \bmod R[0]$
for $j = 3$ **to** b **do**
  $R[j] \leftarrow R[j-1] \cdot A \bmod R[0]$
end
$i \leftarrow \lfloor \log_b d^* \rfloor$
$A \leftarrow 2R[d_i^*] + r_3 \cdot R[0]$
$R[0]^- \leftarrow 2^{\omega-1} \cdot R[0]$
$A \leftarrow A \bmod R[0]^-$
$A \leftarrow A^2 \bmod R[0]^-$
$i \leftarrow i - 1$
while $(i \geq 0)$ **do**
  $A \leftarrow A^{2^{k-1}} \bmod R[0]^-$
  $A \leftarrow A \cdot R[d_i^*] \bmod R[0]^-$
  $A \leftarrow A + A \bmod R[0]^-$
  $A \leftarrow A^2 \bmod R[0]^-$
  $i \leftarrow i - 1$
end
$A \leftarrow 2r_1 \cdot A \cdot R[1] \bmod R[0]^-$
$A \leftarrow A/r_1$
return A

5 COMPARISON WITH OPENSSL

The algorithm used in OpenSSL[1] for the constant time implementation of a modular exponentiation is the classical 2^5-ary exponentiation algorithm, and uses Montgomery multiplication. Each $M^i \bmod N$, for $0 \leq i < 2^k$, are computed and stored in their Montgomery representation. This uses more memory than the proposed algorithm as the modulus cannot be stored in the same memory.

To make the cache accesses behave in a deterministic manner for all possible values of d, the values of $M^i \bmod N$, for $0 \leq i < 2^k$, are mapped so that the choice of any arbitrary $M^i \bmod N$ will access the same cache lines. This is achieved by selecting 2^k to be the same as the number of bytes available in each cache line. One cache line can then be used to store one byte of each $M^i \bmod N$, for $0 \leq i < 2^k$, i.e. if we consider a cache to be a matrix of bytes, where the number columns is the cache line size, each $M^i \bmod N$ is stored in column $i+1$.

No timing analysis can be conducted based on the use of the cache as the same number of cache lines will be accessed for each loop of the algorithm. It also prevents trace-based cache analysis as the same cache lines will be accessed for all possible values of the private exponent. This requires careful implementation, as it is important that the same byte from each $M^i \bmod N$, for $0 \leq i < 2^k$, is stored on the same cache

[1] At the time of writing the most recent release of OpenSSL was version 0.9.8e.

line.

If someone were to take the OpenSSL source and compile it on a platform with a non-standard cache line size (the default in OpenSSL is 32 bytes, and the classical 2^5-ary exponentiation algorithm), without modifying the source, there could be some potential security problems. If, for example, this was implemented on a platform with a cache line size of 16 bytes, then the first cache line would contain the first byte of each each $M^i \bmod N$, for $0 \leq i < 2^4$, and the second cache line would contain the the first byte of $M^i \bmod N$, for $2^4 \leq i < 2^5$. This pattern continues for the bytes stored in the following cache lines. If an attacker is able to determine which set of cache lines are used for each multiplication (i.e. odd or even numbered cache lines) some bits of d can be determined. More precisely, an attacker would be able to determine the most significant bit of each window of k bits.

This problem can be avoided by using the algorithm proposed in this paper, as an attacker may be able to determine some bits of d^* but this will not provide any information on d. However, in an implementation of the proposed algorithm it is still necessary to use the memory mapping described above, so that the same cache lines are accessed for each $M^i \bmod N$, for $0 \leq i < 2^k$. Otherwise a trace-based cache analysis can potentially reveal d^*, which is equivalent to d when used as an exponent modulo N.

This paper does not claim that this represents a security flaw in the current implementation of OpenSSL. Indeed, the use of 16-byte cache lines is considered in the source, but requires the cache line size to be declared. Not all programmers would be aware of the security issues surrounding micro-architectural attacks.

The default implementation of RSA in OpenSSL uses the blinding scheme given in (Chaum, 1985), and described in Algorithm 3. The proposed algorithm will provide a more efficient implementation, as Algorithm 3 requires that $t^e \bmod N$ and $t^{-1} \bmod N$ are stored in memory and periodically updated. Moreover, each time a new t is required a modular inverse needs to be calculated which will increase the time required to compute Algorithm 3.

The proposed algorithm will also provide a more secure implementation, since the exponent is randomised. The appendix describes a theoretical attack that could break the current implementation of OpenSSL, where Algorithm 3 is used, but would not be able to break the proposed algorithm. This is possible because an attacker is required to derive the entire value of d^* in one attack to break the proposed algorithm. In the current implementation of OpenSSL, the repeated use of the same value of d could allow information on different bits of d to be derived from separate attacks.

Algorithm 3: Chaum's blinding scheme

Input: M, d, e where $e \cdot d \equiv 1 \pmod{\phi(N)}$, N, a random value t where $0 \leq t \leq N-1$ and is coprime to N.

Output: $S = M^d \bmod N$.

$A \leftarrow M \cdot t^e \bmod N$
$A \leftarrow A^d \bmod N$
$A \leftarrow t^{-1} \cdot A \bmod N$

return A

6 CONCLUSION

This paper presents a side-channel resistant version of the 2^k-ary exponentiation algorithm for calculating a modular exponentiation. This algorithm is presented in Algorithm 1, and an optimised version is presented in Algorithm 2.

In summary the advantages of the proposed algorithm over the default settings of the implementation used in OpenSSL are:

1. The proposed algorithm requires less memory than the current implementation of OpenSSL as the modulus N can be stored in the same memory as $M^0 \bmod N$. It is also not necessary to store a pair $t^e \bmod N$ and $t^{-1} \bmod N$ in memory, as smaller random values can be used that only have mild constraints. Moreover, it is shown in (Acıiçmez et al., 2007a) that the calculation of the modular inverse necessary for this blinding method could be vulnerable to side-channel analysis.

2. If the source is compiled by a naïve programmer there is less chance of a bug compromising the security of the exponentiation algorithm. An example of this is given in Section 5.

3. The proposed algorithm is more secure against other attacks than the current implementation of OpenSSL. A theoretical attack is described in the appendix that could compromise the security of the current implementation of OpenSSL, even when the current blinding scheme is considered. The proposed algorithm cannot be attacked in this manner.

REFERENCES

Acıiçmez, O. (2007). Private communication.

Acıiçmez, O. and Koç, C. K. (2006). Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138. http://eprint.iacr.org/2006/138/ .

Acıiçmez, O., Gueron, S., and Seifert, J.-P. (2007). New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. Cryptology ePrint Archive, Report 2007/039, 2007, http://eprint.iacr.org/ .

Acıiçmez, O., Koç, C. K., and Seifert, J.-P. (2007a). On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006, http://eprint.iacr.org/ .

Acıiçmez, O., Koç, C. K., and Seifert, J.-P. (2007b). Predicting secret keys via branch prediction. In *Topics in Cryptology — CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag.

Bao, F., Deng, R. H., Han, Y., Jeng, A., Narasimhalu, A. D., and Ngair, T. (1997). Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Security Protocols*, volume 1361 of *Lecture Notes in Computer Science*, pages 115–124. Springer-Verlag.

Barrett, P. (1987). Implementing the Rivest-Shamir-Adleman public-key encryption algorithm on a standard digital processor. In *Advances in Cryptology — CRYPT0 '87*, volume 267 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag.

Bernstein, D. J. (2005). Cache timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf .

Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., and Palermo, G. (2005). AES power attack based on induced cache miss and countermeasures. In *International Symposium on Information Technology: Coding and Computing — ITCC 2005*, pages 586–591. IEEE Computer Society.

Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. In *12th USENIX Security Symposium*, pages 1–14.

Chaum, D. (1985). Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044.

Hachez, G. and Quisquater, J.-J. (2000). Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 293–301. Springer-Verlag.

Joye, M., Quisquater, J.-J., Bao, F., and Deng, R. H. (1997). RSA-type signatures in the presence of transient faults. In *Cryptography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 155–160. Springer-Verlag.

Knuth, D. (2001). *The Art of Computer Programming*, volume 2, Seminumerical Algorithms. Addison–Wesley, third edition.

Kocher, P. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag.

Montgomery, P. (1985). Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521.

OpenSSL (2007). Open source toolkit for SSL/TLS. http://www.openssl.org .

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology — CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag.

Page, D. (2002). Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169. http://eprint.iacr.org/2002/169/ .

Quisquater, J.-J. (1992). Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system. U.S. Patent Number 5,166,978. Also presented at the rump session of EUROCRYPT '90.

Rivest, R., Shamir, A., and Adleman, L. M. (1978). Method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Schindler, W. (2000). A timing attack against RSA with the Chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems —*

CHES 2000, volume 1965 of *Lecture Notes in Computer Science*, pages 109–124. Springer-Verlag.

Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H. (2003). Cryptanalysis of DES implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems — CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag.

Walter, C. D. (1999a). Montgomery exponentiation needs no final subtractions. *Electronic Letters*, 35(21):1831–1832.

Walter, C. D. (1999b). Montgomery's multiplication technique: How to make it smaller and faster. In *Cryptographic Hardware and Embedded Systems — CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 80–93. Springer-Verlag.

APPENDIX

In this appendix a theoretical attack on the current version of OpenSSL is described. The attack assumes that an attacking process is running concurrently with the exponentiation algorithm, that can read and modify arbitrary addresses in RAM. If this process is able to modify the values of $M^i \bmod N$, for $0 \leq i < 2^k$, before they are used to calculate a modular exponentiation an attack can be envisaged based on (Bao et al., 1997; Joye et al., 1997).

An attacker can, arbitrarily, choose some $M^i \bmod N$, for $1 \leq i < 2^k$, and overwrite this value in memory with $M^0 \bmod N$. This has the effect of replacing all b-digits whose value is i with zero (note that $b = 2^k$). An attacker can then seek to determine how many digits were changed from i to zero.

If, for example, the j-th and k-th b-digits of d are changed from i to zero, then the expected signature S' from a message M will satisfy the following equation:

$$\frac{S'^e}{M} \equiv (M^e)^{-(i \cdot b^j)} \cdot (M^e)^{-(i \cdot b^k)} \pmod{N} \quad (\dagger)$$

where e is the public exponent. A more complex equivalence can be determined where an attacker has set a chosen $M^i \bmod N$ to $M^0 \bmod N$, since more digits will be changed than are considered in the above example.

If, for a chosen i, each instance where the b-digit is equal to i is replaced with zero, this could, potentially, allow an attacker to determine where in d each b-digit is equal to i. This could be achieved by calculating the result of $S'^e/M \bmod N$ for all of the possible combinations of changed digits.

For example, if we consider RSA signature generation using a 1024-bit modulus calculated using the 2^5-ary modular exponentiation algorithm (as currently used in OpenSSL). There will be $\lceil 1024/5 \rceil = 205$ loops in the modular exponentiation algorithm. If, for an arbitrary i (for $1 \leq i < b$), $M^i \bmod N$ is changed to $M^0 \bmod N$, this will, statistically, be expected to affect $\lceil 1024/5 \rceil / 2^5 = 6.4$ loops, i.e. on average 6.4 b-digits, that are normally equal to i, will be set to zero. In order to determine which groups of five bits an equation similar to Equation (†) can be determined for each of the $\binom{205}{7} = 2^{41.3}$ possible combinations that cover the expected number of groups of five bits that have changed.

This is likely to be computationally infeasible because of the number of possible changes in d, each of which require the generation of the result of $S'^e/M \bmod N$. However, this expected number of signatures can be significantly reduced if an attacker is able to divide this process into stages, i.e. make a change half way through the modular exponentiation and derive some information, and then repeat the attack and make a change before the exponentiation to complete the attack for a given value of i.

This attack is still valid if the blinding scheme described in Algorithm 3 is used, as an attacker can overwrite some arbitrary $M^i \cdot t^e \bmod N$ with a value equivalent to 1 mod N. No knowledge of t is required since d and N are not modified during the blinding scheme.

This problem can be avoided by using the algorithm proposed in this paper. The attack is still valid, but an attacker will only be able to determine some bits of one instance of d^* and this will not provide any information on d.