

Manger’s Attack revisited†

Falko Strenzke¹

¹ FlexSecure GmbH, Germany **,
strenzke@flexsecure.de

² Cryptography and Computeralgebra, Department of Computer Science,
Technische Universität Darmstadt, Germany

Abstract. In this work we examine a number of different open source implementations of the RSA Optimal Asymmetric Encryption Padding (OAEP) and generally RSA with respect to the message-aimed timing attack introduced by James Manger in CRYPTO 2001. We show the shortcomings concerning the countermeasures in two libraries for personal computers, and address potential flaws in previously proposed countermeasures. Furthermore, we point out a new source of timing differences that has not been addressed previously. We also investigate a new class of related problems in the multi-precision integer arithmetic that in principle allows a variant of Manger’s attack to be launched against RSA implementations on 8-bit and possibly 16-bit platforms.

Keywords: public key encryption scheme, RSA, RSA-OAEP, timing attack, side channel attack

1 Introduction

The widely used RSA public key encryption scheme was found to be insecure [1] when used with PKCS#1 v1.5 encoding [2]. The OAEP encoding [3] was introduced to overcome these security problems. While being formally secure, a straightforward implementation of the RSA-OAEP decoding was found to be vulnerable with respect to timing or fault attacks by James Manger [4]. The current RSA-OAEP specification [3] accounts for these vulnerabilities, and proposes countermeasures.

In this work, we examine two prominent open source cryptographic libraries with respect to the realization of appropriate timing attack countermeasures against Manger’s attack. Specifically, these are the Botan [5] and OpenSSL [6] libraries. We find shortcomings in both implementations concerning the realization of the countermeasures within the RSA-OAEP decoding routine. Furthermore, we show that both libraries also feature another source of timing differences that in principle allows Manger’s attack even when the RSA-OAEP decoding routine is perfectly secured, also enabling attacks against RSA independently of the encoding method. We outline a countermeasure against this new vulnerability. The vulnerability discovered in the OAEP decoding routine of the Botan library was

** A part of the work of F. Strenzke was done at²

† To appear in ICICS 2010. The original publication is available at
www.springerlink.com

fixed in the 1.9.8 and 1.8.9 versions by the library maintainer Jack Lloyd after being informed by us.

Furthermore, we show that an even deeper source of timing differences in the multi-precision integer arithmetic can in principle be used by an attacker to mount Manger's attack especially on 8-bit architectures. Again, this vulnerability is independent of the OAEP decoding method.

2 Preliminaries: RSA-OAEP

In the following, we will very briefly recapitulate the well known RSA public key encryption scheme and then explain the OAEP encoding.

The RSA private key consists of the private exponent d , the public key is given by the modulus n and the public exponent e . RSA encryption is performed by computing the ciphertext $c = m^e \bmod n$, which is decrypted as $m = c^d \bmod n$. The knowledge of the prime factors p, q with $pq = n$ is what enables the holder of the secret key to determine the correct private exponent d .

In [1] it is shown that RSA used with the PKCS#1 v1.5 encoding [2] is vulnerable to certain reaction attacks where the attacker manipulates a target ciphertext he wishes to decrypt, and having access to a decryption oracle is able to observe whether this manipulated ciphertext can be decrypted correctly. To thwart these attacks, RSA-OAEP [3] was introduced. The aim of this so called conversion is to detect any manipulation of the ciphertext during the decryption phase and to refuse to output the decryption result.

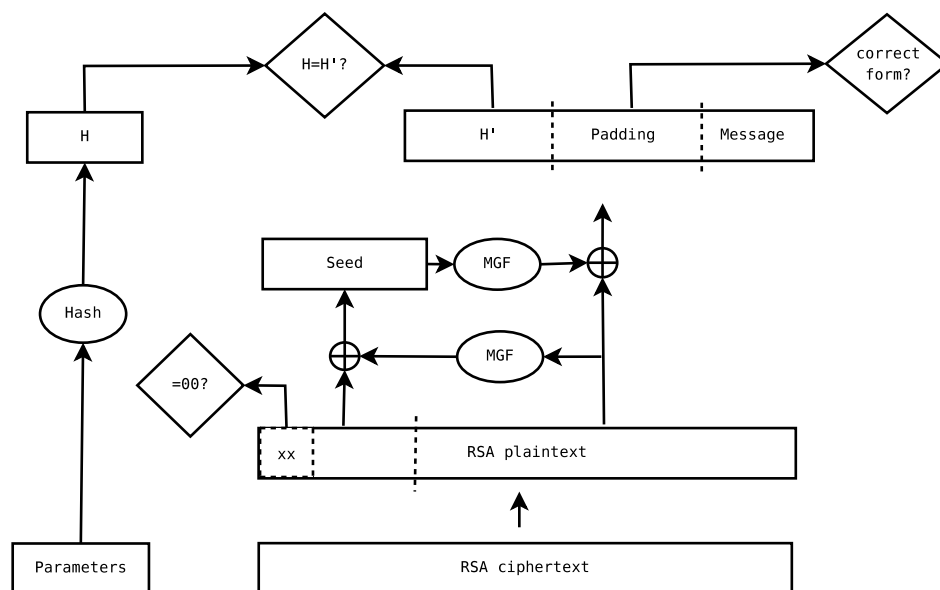


Figure 1: The RSA-OAEP decoding procedure. Here, \oplus denotes XOR.

The RSA-OAEP decryption, depicted in Fig. 1, starts with the RSA decryption. The resulting integer is encoded as an octet string, which is divided into two parts: starting at the most significant octet of the integer, we find a value used for later unmasking of a seed. The RSA-OAEP specification [3] demands that the leading octet of this octet string, denoted by “xx” in the figure, has value zero. This requirement guarantees that the integer representing the message is smaller than the RSA modulus when performing RSA-OAEP encryption. This relation in turn is a requirement of the RSA encryption algorithm. The RSA-OAEP specification [3] demands that after the RSA decryption it is verified that this condition is met and that an error is output in case of a violation. In the remainder of the paper, we will refer to this condition as the first error condition of the OAEP decoding, and will speak of a supernumerary octet in case this error condition is met.

Fig. 1 also shows the further processing of the RSA-OAEP decryption operation. It involves a mask generation function (MGF), specified in [3], which takes a variable length octet string as input and outputs an octet string of fixed length. After the final XOR operation, an octet string consisting of three elements is recovered (the uppermost rectangular box in the figure). The first element, H' (called “pHash” in [3]) is the hash value of an octet string, referred to as the parameters, which must be available to both the encrypting and the decrypting party involved. The padding part is of variable length and has to meet certain requirements for a regular ciphertext, which are of no interest for our analysis.

What is important for our analysis, is that a manipulated ciphertext will always cause at least the comparison $H = H'$ to fail, which we will refer to in the following as the second error condition. The first error condition, i.e. the check whether the highest octet of the RSA plaintext equals 00, will not be triggered in all cases. This is simply due to the fact that the probability for the highest octet being 00 is at least $1/256$ when decrypting random RSA ciphertexts. The third check concerning the form of the padding, is of no importance for the vulnerabilities discussed in this work. Even if this check was executed before the check $H = H'$, there is no known attack based on this error condition.

3 The known Attacks against RSA-OAEP

In [4], it is shown how fault and timing attacks may be possible, if the implementation of the scheme does not include appropriate countermeasures. The attack exploits the fact that in the RSA-OAEP decryption, errors can occur at two different points in the algorithm, as explained in the preceding section. If an attacker can distinguish between these error conditions, he can conduct an adaptively chosen ciphertext attack with the aim of recovering the message corresponding to a certain ciphertext. This is explained in detail in [4]. In the following we only give a brief outline of the principle of this attack, focusing on the underlying information leak.

The attacker performs the attack by creating manipulated versions of the ciphertext he wishes to decrypt. Specifically, given the ciphertext c_0 , he chooses an integer f and computes $c'_0 = f^e c_0 \bmod n$ where e and n are the public

exponent and modulus of the key that was used to encrypt c_0 . The attack builds on the ability of the attacker to let the decryption device decrypt the manipulated cipher text c'_0 and learn whether $m'_0 = fm_0 = c'^d_0$ has a supernumerary octet in the sense described above. Let B be the smallest value of a message m which features a supernumerary octet, then the information learned by the attacker is whether $fm_0 \bmod n \geq B$ is true or false. Repeating this with f chosen based on previous outcomes, he can by and by narrow down the number of possible values of m_0 . This is done with a specific strategy described in [4]. The details of this attack are not necessary to understand the remainder of the paper, the only important thing is the source of the information gain.

The way in which an attacker can learn whether the decryption of a certain ciphertext c_0 caused the supernumerary octet is described as twofold in [4]. First, if the error message in this case is different from the error message that occurs when checking whether $H = H'$, and the attacker has access to these error messages, the attack can be mounted as a fault attack. But even if the error messages are indistinguishable, there is still a chance for the attacker to distinguish at which stage in the algorithm the error was caused based on the running time [4]. The attack becomes especially dangerous if the computation of the parameter hash value is performed after the check of the first error condition and the attacker is able to provide parameters of arbitrary length³. Then the timing differences based on the first error condition can become enormous.

4 Analysis of two open Source RSA-OAEP Implementations

In this section, we analyze the implementation of the RSA-OAEP decoding in the open source cryptographic libraries Botan [5] and OpenSSL [6] concerning their defense against Manger's attack [4].

In advance, we wish to point out that the most dangerous timing attacks, where the attacker is able to control the size of the parameters to be hashed, are not possible for RSA-OAEP as it is implemented in Botan and OpenSSL. This is due to the fact that in Botan the parameters have a preset value which is the empty string, and in OpenSSL, the OAEP decoding is continued even in case of the fulfillment of the first error condition, thus ensuring against conditionally performing the computation of the parameter hash.

4.1 The RSA-OAEP Decoding Operation in OpenSSL

In OpenSSL 1.0.0, the implementation of RSA-OAEP decoding is found in the file `rsa/rsa_oaep.c` in the function `RSA_padding_check_PKCS1_OAEP()`. In List. 1, we show the implementation of the countermeasure against Manger's attack within this function. In Line 109 the difference between the actual number of non-zero octets of the plaintext octet string (`flen`) and the maximal allowed

³ Only limited by the input size limit of the employed hash function, which is 2^{64} bits for SHA-1, for instance.

length (`num`) is computed. The error condition is checked in the if-statement in the subsequent line. Obviously, the implementer did not consider it a problem that in case of the fulfillment of the first error condition, the code inside the if-block causes a timing difference compared to the case where the condition is not fulfilled. As a consequence, it cannot be excluded that the conditional branching can be detected through the timing in certain scenarios. Furthermore, based on the conditional branching, the attack can in principle be mounted as a branch prediction attack [7].

Basically, it is easily possible to remove this vulnerability by using branch free code employing the techniques shown in [8, 9]. But since implementing the countermeasure proposed by us in Sec. 5.2 entirely removes the necessity to deal with whole issue arising in the context of the supernumerary octet, we do not address such a solution here.

```

109  lzero = num - flen;
110  if (lzero < 0)
111    {
112      /* signalling this error immediately after detection might allow
113       * for side-channel attacks (e.g. timing if 'plen' is huge
114       * -- cf. James H. Manger, "A Chosen Ciphertext Attack on RSA
115        *   Optimal
116       * Asymmetric Encryption Padding (OAEP) [...]", CRYPTO 2001),
117       * so we use a 'bad' flag */
118      bad = 1;
119      lzero = 0;
120      flen = num; /* don't overflow the memcopy to padded_from */
121    }

```

Listing 1: The implementation of the countermeasure against Manger's attack in the RSA-OAEP decoding routine of OpenSSL-1.0.0.

4.2 The RSA-OAEP Decoding Operation in Botan

The decoding operation in Botan-1.9.7 is found in the file `pk_pad/eme1/eme1.cpp`, in the function `EME1::unpad()`. The beginning of the function is shown in List. 2. The implementation is not vulnerable to fault attacks, since exactly the same exception with the same error message is thrown for all possible errors occurring during the decoding operation. Timing attack countermeasures are not included. With respect to the implementation of corrective countermeasures in this function, the same considerations as given for OpenSSL in Sec. 4.1 apply.

4.3 Potential Risks of previously proposed Countermeasures

As mentioned in Sec. 4.1, we aim at a more fundamental countermeasure that takes effect already before the OAEP decoding routine, this will be discussed in Sec. 5. But since in [8],[4] and [3] countermeasures to be implemented within

```

55     key_length /= 8;
56     if(in_length > key_length)
57         throw Decoding_Error("Invalid_OAEP_encoding");
58
59     SecureVector<byte> tmp(key_length);
60     tmp.copy(key_length - in_length, in, in_length);
61
62     mgf->mask(tmp + HASH_LENGTH, tmp.size() - HASH_LENGTH, tmp,
63             HASH_LENGTH);
64     mgf->mask(tmp, HASH_LENGTH, tmp + HASH_LENGTH, tmp.size() -
65             HASH_LENGTH);
66
67     for(u32bit j = 0; j != Phash.size(); ++j)
68         if(tmp[j+HASH_LENGTH] != Phash[j])
69             throw Decoding_Error("Invalid_OAEP_encoding");

```

Listing 2: The implementation of the RSA-OAEP decoding routine of Botan-1.9.7.

the OAEP decoding have been proposed, we wish to point out the possibility of creating new power analysis vulnerabilities when following these propositions.

In [8], the authors suggest to react to a supernumerary octet in the following manner: if the first error condition in the OAEP decoding is fulfilled, one shall “generate a dummy value (which can be selected arbitrarily from the domain of OAEPDECODE)”, where the dummy value shall be used as the decoded RSA plaintext in the further processing of the OAEP decoding. While this statement is not entirely clear about whether to use random values generated anew whenever the error condition is fulfilled, it could at least be interpreted in this way. At this point, [4] explicitly suggests to use random values. This, however, would be a problem: it would potentially reveal the error condition by introducing a certain amount of randomization in the further OAEP decoding procedure, which is presumably entirely deterministic if the first error condition is not fulfilled. It might thus be possible to detect the first error condition by repeatedly letting the device decrypt the target ciphertext and computing the variances of the set of power consumption samples at the corresponding points in time: the randomization should be revealed by a larger variance than for a ciphertext which does not lead to the fulfillment of this error condition.

Similarly, in [3], it is suggested that if the first error condition is fulfilled one shall “proceed to step 5 with EM set to a string of zero octets”, where “EM” refers to the decoded RSA plaintext and “step 5” refers to the remaining computations of the OAEP decoding. This is also not a good advice, since a string of all zeros is an extreme case of low hamming weight. Knowing that differences in hamming weights during a computation are the most common targets of power analysis attacks [10], it seems highly likely that an attacker will be able to deduce the fulfillment of the error condition from the power trace.

Instead, if one decides to implement a countermeasure within the OAEP decoding routine, one should follow the principle of least modification: simply ignore the supernumerary octet without introducing a timing difference. In this case, the second error condition will be fulfilled anyway, ensuring the overall correct decryption result, i.e. the indication of “decryption error”. This is because, as is immediately obvious, the check for the first error condition is not contributing to the security of the scheme since Manger’s attack builds on frequently bypassing it.

Naturally, the above considerations only apply to the case where power analysis attacks are feasible.

5 A new Vulnerability in the Integer to Octet String Conversion

While the vulnerabilities in the previous section stem from insufficient countermeasures against Manger’s attack, we will show in this section, that even given a perfectly secured OAEP decoding routine there are potential vulnerabilities already in the integer to octet string conversion preceding the OAEP decoding step. We will then discuss appropriate countermeasures.

5.1 The Integer to Octet String Conversion in OpenSSL and Botan

Concerning the implementation of the integer encoding routine, OpenSSL and Botan both share the vulnerability that the running time of this routine linearly depends on the number of octets that are needed to represent the integer. Thus, if in a given scenario an attacker is able to detect these timing differences, he would be able to mount Manger’s attack based on this side channel information.

In List. 3 we show the routine used for the integer encoding in Botan-1.9.7, located in the file `math/bigint/bigint.cpp`. First, in line 337, the variable `sig_bytes` is assigned the number of bytes the resulting octet string will consist of. Then, in the subsequent line, a loop is started that has as many iterations as the value of `sig_bytes`, obviously causing the timing dependency mentioned above. In OpenSSL, the integer encoding is done in the function `BN_bn2bin()` in the file `bn/bn_lib.c`. It is fully equivalent to the Botan implementation, so we omit the analysis of that function and simply record that OpenSSL’s integer encoding routine suffers from the same vulnerability as Botan’s.

```

335 void BigInt::binary_encode(byte output[]) const
336 {
337     const u32bit sig_bytes = bytes();
338     for(u32bit j = 0; j != sig_bytes; ++j)
339         output[sig_bytes-j-1] = byte_at(j);
340 }
```

Listing 3: The implementation of the integer encoding in Botan-1.9.7.

5.2 The Solution: no Secret dependent Branching

The solution for both implementations of RSA-OAEP considered in this work is to use a modified integer encoding routine. This routine has to satisfy two requirements: Firstly, it should receive the maximum number of octets allowed by RSA-OAEP as a function parameter, called S_{\max} from here on, and discard all octets that exceed this maximal size. Secondly, it should have the same running time independently of whether the integer's natural encoded value comprises S_{\max} or $S_{\max} + 1$ octets. The first requirement removes the need to check for the input size in the OAEP decoding operation, since now it is guaranteed that the maximal size is not exceeded already during the integer encoding. The second requirement makes sure that not even a “tiny” revealing timing difference occurs during the integer encoding.

In order to achieve the goal of secret independent running time, we have to avoid secret based branching in the routine. To this end, techniques similar to those proposed in [8, 9] should be used. Those techniques are based on replacing conditional statements with logical masking. Furthermore, in order to avoid basically any possibility of the compiler introducing conditional branching where it is not desired, one should avoid any use of comparative statements when dealing with secrets. The reason is that compilers might implement these comparisons with conditional branching, as is pointed out in [8]. We thus recommend to use only logical masking in the implementation of the countermeasure and avoid the use of comparison operators. For the generation of the masks using only logical operations see for instance the example given in [11]. Furthermore, we recommend the use of the `volatile` specifier, that is part of the C programming language specification. Declaring a variable in this way tells the compiler that it might be changed asynchronously by another process or thread (as it would for instance be the case when using shared memory). This removes the compilers freedom of optimizing code involving the variable [12]. Declaring our logical mask variables in this way, we render it highly unlikely that the compiler transforms the logical operations into code containing conditional branching.

In App. A we give C++ code employing all the mentioned features to realize the timing attack secure integer to octet string conversion for the Botan library.

6 New Vulnerabilities in the Multi-Precision Integer Arithmetic

In the previous sections we have seen that prominent implementations of RSA-OAEP are not entirely secured against Manger's attack, and that the integer encoding routines feature timing attack vulnerabilities that have not been investigated so far. In this section we will show that timing differences based on the number of leading zero bytes of the plaintext can also appear already within the RSA computation itself. This, however, seems only possible on 8-bit or 16-bit platforms as we will show in the following.

Let us assume a particular implementation of RSA that uses so called base blinding [13] which is a well known countermeasure against timing and power analysis attacks against the RSA private key. It is shown in Alg. 1.

Algorithm 1 RSA decryption with base blinding side channel countermeasure

Require: RSA ciphertext c , modulus n , public exponent e and private exponent d
Ensure: RSA plaintext m

$$r \leftarrow \text{random number}$$

$$c' \leftarrow r^e c \bmod n$$

$$m' \leftarrow c'^d \bmod n$$

$$m \leftarrow m' r^{-1} \bmod n$$

Obviously, the last operation that leads to the recovering of m is a modular reduction modulo n , which we assume to be implemented as a multi-precision integer division. We will consider alternatives in the subsequent general discussion. The potential vulnerability we wish to point out will most probably be present in any straightforward multi-precision integer implementation. But to ease the discussion, we will turn to a concrete implementation example first, and afterwards generalize the results.

6.1 The Example of PolarSSL

Since the vulnerability we are going to discover will only be found on 8-bit or 16-bit platforms, we choose a cryptographic library that is intended for the use on embedded platforms, namely the PolarSSL library [14]⁴.

In order to understand the vulnerability we first need to understand how multi-precision integers are implemented in PolarSSL. A multi-precision integer in PolarSSL is realized as an object that contains a pointer \mathbf{p} to an array of words representing the integer and a native integer \mathbf{n} indicating the number of words allocated in that array. Independently of the specific implementation of the division routine the integer m is found as the last remainder that occurs during the division. The multi-precision integer object representing the last remainder in the division routine of PolarSSL, which goes by the name `mpi_div_mpi()`, is a local variable and thus is not the one that is returned by the function as the result. The multi-precision integer returned by the function is in fact assigned the value of the local remainder, this is done with the help of the function `mpi_copy()`, shown in List. 4. In this function we see a for-loop running through $\mathbf{Y} \rightarrow \mathbf{p}$ from the highest word of the multi-precision integer \mathbf{Y} serving as the source, stopping once the leading zero words have been consumed. From this point on, the variable \mathbf{i} carries the number of significant words of \mathbf{Y} . A call to `mpi_grow()` then ensures that $\mathbf{X} \rightarrow \mathbf{p}$ is large enough to hold the contents of the origin. Consequently, with a call to `memset()` the contents of \mathbf{X} are cleared, and via a call to `memcpy()` the significant words of \mathbf{Y} are copied to \mathbf{X} .

It is immediately clear that the size parameter in the call to `memcpy()` is the number of significant 8-bit words of \mathbf{Y} , i.e. the integer representing the message. Note that `ciL` is a compile time constant that equals one when 8-bit words are used inside `mpi`. Since the running time of the `memcpy()` implementation can

⁴ Note, however, that in this library RSA is not actually implemented with base blinding, this is an additional assumption for our analysis.

generally be assumed to be dependent on the size parameter, we obviously have found a new source for timing differences based on the number of octets needed to represent the message.

The remaining operations inside the `mpi_copy()` routine are also prone of having associated running times related to the number of 8-bit words in Y , but all of them are more or less dependent on the number of leading zero words in the allocated arrays of X and Y , which in turn depend on their “history”, which we have not analyzed here⁵. Clearly, one has to be aware of the fact that though there might also be effects that decrease the running time based on the number of significant 8-bit words in Y , it cannot be safely assumed that this results in a total compensation. This could only happen by chance on specific platforms. Of course, also the net timing effect will clearly be platform dependent.

```

129 int mpi_copy( mpi *X, const mpi *Y )
130 {
131     int ret, i;
132     if( X == Y )
133         return( 0 );
134     for( i = Y->n - 1; i > 0; i-- )
135         if( Y->p[i] != 0 )
136             break;
137     i++;
138     X->s = Y->s;
139     MPI_CHK( mpi_grow( X, i ) );
140     memset( X->p, 0, X->n * ciL );
141     memcpy( X->p, Y->p, i * ciL );
142 cleanup:
143     return( ret );
144 }
```

Listing 4: The function `mpi_copy()` of the PolarSSL 0.13.1 library. It copies the contents of Y to X .

6.2 Generalization of the Vulnerability

We wish to point out that the basic principle of this type of vulnerability is much more general than the concrete example analyzed above. Regardless of how the RSA decryption and the multi-precision integer arithmetic are actually implemented, it is always likely that the last routine dealing with the integer m counts the leading zero words in order to set the length appropriately or that only the significant words are copied, be it in a division, subtraction, Montgomery multiplication, or a copy/assignment routine as in the example above. The chances

⁵ Remember that this analysis basically serves as a case study to show a rather general type of problem, thus we are not so much interested in the more arbitrary features of the PolarSSL implementation.

for the attacker to be actually able to exploit this vulnerability, however, will certainly depend on implementation details and the platform.

Also note that the assumption that base blinding according to Alg. 1 is used in the implementation has the consequence that the system becomes easier to attack: while in a totally unsecured implementation of the RSA decryption other timing differences during the whole decryption operation will render the detection of the supernumerary word very difficult or impossible, the base blinding randomizes this process. Thus, by repeatedly letting the device decrypt the same manipulated ciphertext multiple times and averaging over the associated timings this noise will be reduced.

As a consequence, even given constant time integer encoding and a secure implementation of the OAEP decoding routine, there are sources of timing differences in the multi-precision integer arithmetic that potentially reveal a leading zero byte. This is the case when the word size used by the multi-precision integer implementation is 8-bit. In the case of 16-bit words, Manger's attack would have to be slightly modified, i.e. the queries would then reveal two leading zero octets instead of one, where it is unclear in what extent this affects the practicability of the attack. For implementations using 32-bit words, however, the probability of four leading zero octets must be assumed to be far too low to enable practical attacks⁶. Obviously, this vulnerability, like the one discussed in Sec. 5, is not bound to the OAEP encoding method.

7 Conclusion

In the author's opinion, the results of this work suggest that a systematic approach to identifying all sources of possible timing differences has to be pursued for each known principal side channel based information gain. In the case of Manger's attack, this doesn't seem to have taken place even though the basic problem is known for almost 10 years now. Clearly, we cannot claim that the outlined problems imply exploitable vulnerabilities on all platforms. The level of analysis pursued in this work is that of principle, theoretic sources of timing differences enabling an information gain by an attacker. But the presence of these principle problems forces a developer or user of systems employing implementations of these cryptographic functions to verify that they are not vulnerable on the specific platform and in the specific environment he uses.

Furthermore, especially the implementation of the countermeasure against Manger's attack in OpenSSL (Sec. 4.1) shows that a common notion about the relevance of secret related timing differences cannot be assumed. The implementer obviously simply took for granted that the timing differences introduced by the if-statement do not matter on the platforms and in the environment where OpenSSL's RSA-OAEP decoding will be used. This may even be well justified for the general case, where an attacker has to face noisy timings resulting from

⁶ These assumptions are based on the usual choices for the RSA bit key sizes, that are at least divisible by 32.

modern superscalar CPUs, multi tasking operating systems and network connections with varying delays. However sufficient these assumptions may be for the safety of the countermeasure, the point is, that they are neither made explicit nor verified. The unaddressed problem is that a user of the library will in general not be aware of the potential problems and thus cannot know when he runs into dangers with his specific setup.

References

1. Bleichenbacher, D.: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1. In: CRYPTO, Springer-Verlag (1998) 1–12
2. RSA Data Security, Redwood City, CA: PKCS#1: RSA Encryption Standard (1993) Version 1.5.
3. RSA Laboratories, RSA Security Inc., 20 Crosby Drive, Bedford, MA 01730 USA: RSAES-OAEP Encryption Scheme (2000)
4. Manger, J.: A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS#1 v2.0. CRYPTO (2001)
5. The Botan Library: botan.randombit.net.
6. The OpenSSL Library: www.openssl.org.
7. Aciçmez, O., Çetin Kaya Koç, Seifert, J.P.: Predicting secret keys via branch prediction. In: CT-RSA 2007. (2007) 225–242
8. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In: ICISC. (2005) 156–168
9. Coppens, B., Verbauwhede, I., Bosschere, K.D., Sutter, B.D.: Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. Security and Privacy, IEEE Symposium on (2009) 45–60
10. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer (2007)
11. Strenzke, F., Tews, E., Molter, H.G., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. Post-Quantum Cryptography, LNCS (2008)
12. Software Engineering Institute: <https://buildsecurityin.us-cert.gov/bsi-rules/home/g1/771-BSI.html>.
13. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Proceedings of the 16th Annual International Cryptology Conference Advances in Cryptology, Springer-Verlag (1996) 104–113
14. The PolarSSL Library: <http://www.polarssl.org/>.

Appendix

A Timing Attack resistant Integer to Octet String Conversion

In List. 5 we present a secure version of the integer to octet string conversion for the Botan library. It follows all the recommendations from Sec. 5.2. In order to generally avoid timing related vulnerabilities with respect to the properties of the encoded integer, we design the algorithm in such a way that the running

time is totally independent of the actual number of octets needed to represent the integer.

Please note that the given implementation assumes a two's complement machine, which is a negligible restriction considering the prevalence of this integer representation. List. 6 and 7 show subroutines called from within the secure encoding routine, and are adhering to the same principles.

```

1 SecureVector<byte> BigInt::binary_encode_ta_sec(u32bit max_enc_len)
   const
2 {
3   /* set the number of bytes that the integer would normally need:
   */
4   const u32bit sig_bytes = bytes();
5   volatile u32bit tracker_mask = 0;
6   u32bit act_size = min_ta_sec(sig_bytes, max_enc_len);
7   u32bit offset = max_enc_len - sig_bytes;
8   /* if sig_bytes is larger than max_enc_len, then there shall be
   no offset
   * (which is negative so far)
   */
9   */
10  volatile u32bit offs_mask = offset & (1 << 31); /* is offset
   negative ? */
11  offs_mask = expand_mask_u32bit(offs_mask); /* FF..FF if negative
   */
12  offs_mask = ~offs_mask; /* 00..00 if negative offset */
13  offset = (offset & offs_mask); /* offset >= 0 now */
14
15  SecureVector<byte> result(act_size);
16  for(u32bit j = 0; j != max_enc_len; ++j)
17  {
18    /* zero iff j for the first time is too large for the actual
   bigint: */
19    volatile u32bit mask_left_range = sig_bytes - j;
20    mask_left_range = expand_mask_u32bit(mask_left_range);
21    /* now lives up to its name: */
22    mask_left_range = ~mask_left_range;
23    /* now update tracker_mask to keep track of whether j has
   become too high */
24    tracker_mask |= mask_left_range;
25    /* now make use of the knowledge in tracker_mask: */
26    mask_left_range |= tracker_mask;
27    /* finally access the byte. normal access when in range, when
   not in range,
   * we put a zero. the access into the bigint however will be at
   the
   * beginning of its array. */
28    u32bit result_pos = (max_enc_len-j-1-offset) & ~mask_left_range
29    ;
30    u32bit source_pos = (j & ~mask_left_range) | ((sig_bytes - 1) &
31    mask_left_range);
32    result[result_pos] = ((byte_at(source_pos) & ~mask_left_range))
33    |
34    (result[result_pos] & mask_left_range);
35  }
36  return result;
37 }

```

Listing 5: Constant time integer encoding for the Botan library to be used in the RSA-OAEP decryption routine.

```

1 u32bit min_ta_sec(u32bit a, u32bit b)
2 {
3     u32bit a_larger = b - a; /* negative if a larger */
4     volatile u32bit mask_a_larger = a_larger & (1<<31);
5     mask_a_larger = expand_mask_u32bit(mask_a_larger); /* FF..FF if a
        larger */
6     return (a & ~mask_a_larger) | (b & mask_a_larger);
7 }

```

Listing 6: A function that computes the minimum of two unsigned values with purely logical operations.

```

1 u32bit expand_mask_u32bit(u32bit in)
2 {
3     volatile u32bit result = in;
4     result |= result >> 1;
5     result |= result >> 2;
6     result |= result >> 4;
7     result |= result >> 8;
8     result |= result >> 16;
9     result &= 1;
10    result = ~(result - 1);
11    return result;
12 }

```

Listing 7: A function that expands a mask in the sense that upon in=0 the output is 0x00...00 and 0xFF...FF otherwise, where only logical operations are used.