Efficient Padding Oracle Attacks on Cryptographic Hardware*

Romain Bardou[1], Riccardo Focardi[2]**, Yusuke Kawamoto[3]***,
Lorenzo Simionato[2]†, Graham Steel[4]***, and Joe-Kai Tsay[5]***

[1] INRIA SecSI, LSV, CNRS & ENS-Cachan, France
[2] DAIS, Università Ca' Foscari, Venezia, Italy
[3] School of Computer Science, University of Birmingham, UK
[4] INRIA Project ProSecCo, Paris, France
[5] Department of Telematics, NTNU, Norway

Abstract. We show how to exploit the encrypted key import functions of a variety of different cryptographic devices to reveal the imported key. The attacks are padding oracle attacks, where error messages resulting from incorrectly padded plaintexts are used as a side channel. In the asymmetric encryption case, we modify and improve Bleichenbacher's attack on RSA PKCS#1v1.5 padding, giving new cryptanalysis that allows us to carry out the 'million message attack' in a mean of 49 000 and median of 14 500 oracle calls in the case of cracking an unknown valid ciphertext under a 1024 bit key (the original algorithm takes a mean of 215 000 and a median of 163 000 in the same case). We show how implementation details of certain devices admit an attack that requires only 9 400 operations on average (3 800 median). For the symmetric case, we adapt Vaudenay's CBC attack, which is already highly efficient. We demonstrate the vulnerabilities on a number of commercially available cryptographic devices, including security tokens, smartcards and the Estonian electronic ID card. The attacks are efficient enough to be practical: we give timing details for all the devices found to be vulnerable, showing how our optimisations make a qualitative difference to the practicality of the attack. We give mathematical analysis of the effectiveness of the attacks, extensive empirical results, and a discussion of countermeasures and manufacturer reaction.

1 Introduction

Tamper-resistant cryptographic security devices such as smartcards, USB keys, and Hardware Security Modules (HSMs) are an increasingly common component of distributed systems deployed in insecure environments. Such a device must offer an API to the outside world that allows the keys stored on the device to be used for cryptographic functions and permits key management operations, but without compromising security. The most commonly used standard for designing cryptographic device interfaces, RSA PKCS#11 [28], is known to have vulnerabilities if the attacker is assumed to have access to the full API, and can therefore make attacks by combining commands in unexpected ways [4,5,7]. In this paper, we describe a different way to attack keys stored on the device using only decryption queries performed by a single function, usually the `C_UnwrapKey` function for encrypted key import. These attacks are cryptanalytic rather than purely logical, and hence require multiple command calls to the interface, but the attacker only needs access to one seemingly innocuous command, subverting the typical countermeasure of introducing access control policies permitting only limited access to the interface.

We will show how the `C_UnwrapKey` command from the PKCS#11 API is often implemented on commercially available devices in such a way that it offers a 'padding oracle', i.e. a side channel allowing him to see whether a decryption has succeeded or not. We give two varieties of the attack: the first for when the imported key is encrypted under a public key using RSA PKCS#1 v1.5 padding, which is still by far the most common and often the only available mechanism on the devices we obtained, and the second for when the key is encrypted under a symmetric key using CBC and PKCS#5 padding. The first attack is based on Bleichenbacher's well-known attack [2]. Although commonly known as the 'million message attack', in practice Bleichenbacher's attack requires only about 215 000 oracle calls on average against a 1024 bit modulus when the ciphertext under attack is known to be a valid PKCS#1 v1.5 block. This is however not efficient enough to be practical on low power devices such as smartcards which perform RSA operations rather slowly. We give a modified algorithm which

---

* Full version of a paper with the same title which appears at *CRYPTO'12*.

** Work partially supported by the RAS Project *TESLA: Techniques for Enforcing Security in Languages and Applications*

*** Work partially carried out while at INRIA SecSI, LSV, CNRS & ENS-Cachan, France

† Now at Google Inc.

results in an attack which is 4 times faster on average than the original, with a median attack time over 10 times faster. We also show how the implementation details of some devices can be exploited to create stronger oracles, where our algorithm requires only 9400 mean (3800 median) calls to the oracle. At the heart of our techniques is a small but significant theorem that allows not just multiplication (as in the original attack) but also division to be used to manipulate a PKCS#1 v1.5 ciphertext and learn about the plaintext. In the second attack we use Vaudenay's technique [30] which is already highly efficient. Countermeasures to such chosen ciphertext attacks are well known: one should use an encryption scheme proven to be secure against them. We discuss the availability of such modes in current cryptographic hardware and examine what other countermeasures could be used while such modes are still not available.

In summary, our contributions are the following: i) new results on PKCS#1 v1.5 cryptanalysis that, when combined with the 'parallel threads' technique of Klima-Pokorny-Rosa [29] (which on its own contributes a 38% improvement on mean and 52% on median) results in an improved version of Bleichenbacher's algorithm giving a fourfold (respectively tenfold) improvement in mean (respectively median) attack time compared to the original algorithm (measured over 1000 runs with randomly generated 1024 bit RSA keys and randomly generated conforming plaintexts); ii) demonstration of the attacks on a variety of cryptographic hardware including USB security tokens, smartcards and the Estonian electronic ID card, where we found various implementations of the oracle, and adapted our algorithm to each one, resulting in attacks with as few as 9400 mean (3800 median) oracle calls on the most vulnerable devices; iii) analysis of the complexity of the attacks, empirical data, and manufacturer reaction.

In the next section, we describe the padding attacks relevant to this work and describe our modifications to Bleichenbacher's algorithm. The results on commercial devices are described in section 3. We discuss countermeasures in section 4. Finally we conclude with a discussion of future work in section 5.

2 Padding Oracle Attacks

A padding oracle attack is a particular type of side channel attack where the attacker is assumed to have access to an oracle which returns true just when a chosen ciphertext corresponds to a correctly padded plaintext under a given scheme.

2.1 Bleichenbacher's Attack

Bleichenbacher's padding oracle attack, published in 1998, applies to RSA encryption with PKCS#1 v1.5 padding [2]. Let n, e be an RSA public key and d be the corresponding private key, i.e. $n = pq$ and $ed \equiv 1$ (mod $\phi(n)$). Let k be the byte length of n, so $2^{8(k-1)} \leq n < 2^{8k}$. Suppose we want to encrypt a plaintext block P where P is l bytes long. Under PKCS#1 v1.5 we first generate a pseudorandom non-zero padding string PS which is $k - 3 - l$ bytes long. We allow l to be at most $k - 11$, so there will be at least 8 bytes of padding. The block for encryption is now created as

$$\texttt{0x00}, \texttt{0x02}, PS, \texttt{0x00}, P$$

We call a correctly padded plaintext and a ciphertext that encrypts a correctly padded plaintext *PKCS conforming* or just *conforming*. For the attack, imagine, as above, that the attacker has access to an oracle that tells him just when an encrypted block decrypts to give a conforming plaintext, and assume he is trying to obtain the message $m = c^d \ mod \ n$, where c is an arbitrary integer. He is going to choose integers s, calculate $c' = c \cdot s^e \ mod \ n$ and then send c' to the padding oracle. If c' is conforming then he learns that the first two bytes of $m \cdot s$ are $\texttt{0x00}, \texttt{0x02}$. Hence, if we let $B = 2^{8(k-2)}$, $2B \leq m \cdot s \ mod \ n < 3B$. The idea is to repeat the process for many values of s until only a single plaintext is possible.

2.2 Improving the Bleichenbacher Attack

Let us first review in a little more detail the original attack algorithm. We are trying to obtain message $m = c^d \ mod \ n$ from ciphertext c. In step 1 (Blinding), we search for a random integer value s_0 such that $c(s_0)^e \ mod \ n$ is conforming, by accessing the padding oracle. We let $c_0 = c(s_o)^e \ mod \ n$ and $m_0 = (c_0)^d \ mod \ n$. Note that $m_0 = ms_0 \ mod \ n$. Thus, if we recover m_0 we can compute the target m as $m_0(s_0)^{-1} \ mod \ n$. If the target ciphertext is already conforming, we can set s_0 to 1 and skip this step.

We let $B = 2^{8(k-2)}$. If c_0 is conforming, $2B \leq m_0 < 3B$. Thus, we set the initial set M_0 of possible intervals for the plaintext as $\{[2B, 3B - 1]\}$. In step 2, we search for s_i such that $c(s_i)^e \ mod \ n$ is conforming. In step 3,

we apply the s_i we found to narrow the set of possible intervals M_i containing the value of the plaintext, and in step 4 we either compute the solution or jump back to step 2.

We are interested in improving step 2, i.e. the search for s_i. We give step 2 of the original algorithm below, and omit the other steps (in the appendix we give our modified algorithm, of which step 1.a equals step 1 of the original algorithm, whereas steps 3 and 4 are unchanged from the original).

Step 2a If $i = 1$ (i.e. we are searching for s_1), search for the smallest positive integer $s_1 \geq n/(3B)$ such that $c_0(s_1)^e \; mod \; n$ is conforming. It can be shown that smaller values of s_1 never give a conforming ciphertext.

Step 2b If $i > 1$ and $|M_{i-1}| > 1$, search for the smallest positive integer $s_i > s_{i-1}$ such that $c_0(s_i)^e \; mod \; n$ is conforming.

Step 2c If $i > 1$ and $|M_{i-1}| = 1$, i.e. $M_{i-1} = \{[a, b]\}$, choose small r_i, s_i such that

$$r_i \geq 2\frac{bs_{i-1} - 2B}{n} \quad \text{and} \quad \frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

until $c_0(s_i)^e \; mod \; n$ is conforming. Intuitively, the bounds for s_i derive from the fact that we want $c_0(s_i)^e \; mod \; n$ conforming, i.e. $2B \leq m_0 s_i - r_i n < 3B$, for some r_i, and from the assumption $a \leq m_0 \leq b$. As explained in the original paper, the constraint on r_i aims at dividing the remaining interval in half so to maximize search performance.

Some features of the algorithm's behaviour were already known from the original paper. For example, step 2a/b will in general be executed only very few times (in roughly 90% of our trials, step 2b was executed a maximum of once, and in 32% of cases not at all). However, a lot of the expected calls are here, since each time we just search naïvely for the next s_i, which takes an expected $1/Pr(P)$ calls where $Pr(P)$ is the probability of a random ciphertext decrypting to give a conforming block. Step 2c, meanwhile, is highly efficient, but is only applicable if there is only one interval left. Furthermore it cannot be directly applied to the original interval $\{2B, 3B - 1\}$ (since the bound on r_i, s_i collapses and we end up with the same search as in step 2a). Based on this observation, we devised a new method for narrowing down the initial interval so that 'step 2c-like' reasoning could be applied to speed up the search for s_1.

Trimming M_0 First observe that as well as multiplying the value of the decrypted plaintext ($mod \; n$) by some integer s, we can also divide it by an integer t by multiplying the original ciphertext by $t^{-e} \; mod \; n$. Multiplication modulo n is a group operation on $(\mathbb{Z}_n)^*$, so inverses are unique. If the original plaintext was divisible by t, the result $m_0 \cdot t^{-1} \; mod \; n$ will just be m_0/t, otherwise it will be some other value in the group that we in general cannot predict without knowing m_0. The following holds.

Proposition 1. *Let u and t be two coprime positive integers such that $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$. If m_0 and $m_0 \cdot ut^{-1} \; mod \; n$ are PKCS conforming, then m_0 is divisible by t.*

Proof. We have $m_0 u < m_0 \frac{3}{2} t < 3B \frac{3}{2} t < n$. Thus, $m_0 u \; mod \; n = m_0 u$. Let $x = m_0 \cdot ut^{-1} \; mod \; n$. We know $x < 3B$ since it is conforming. Thus $xt < 3Bt < n$ and $xt \; mod \; n = xt$. Now, $xt = xt \; mod \; n = m_0 u \; mod \; n = m_0 u$ which implies t divides m_0.

By Proposition 1, if we find coprime positive integers u and t, $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$ such that for a PKCS conforming m_0, $m_0 \cdot ut^{-1} \; mod \; n$ is also conforming, then we know that m_0 is divisible by t and $m_0 \cdot ut^{-1} \; mod \; n = m_0 \frac{u}{t}$. As a consequence

$$2B \cdot t/u \leq m_0 < 3B \cdot t/u.$$

Note that since we already know $2B \leq m_0 < 3B$ we can restrict our search to t and u such that $2/3 < u/t < 3/2$. Given a fraction u/t in this range, and assuming m_0 is uniformly distributed in the interval, we can calculate the probability that $m_0 \cdot u/t$ is also in $[2B, 3B - 1]$ by first calculating the probability that t divides m_0 and then the probability that it is in range: for $2/3 < u/t < 1$ it is $1/t \cdot (3 - 2 \cdot t/u)$ and for $1 < u/t < 3/2$ it is $1/t \cdot (3 \cdot t/u - 2)$. Hence it is clear that fractions with smaller denominators are more likely to work, and that u should be as close to t as possible. We apply this by constructing a list of suitable fractions u/t that we call 'trimmers'. In practice, we use a few thousand trimmers and take $t \leq 2^{12}$ as the implementations typically satisfy $n \geq 2^{8k-1}$. For small t we add all suitable u/t to the trimmer list, but for $t > 50$ we add only $(t - 1)/t$ and $(t + 1)/t$. This bound of 50 is a heuristic value found by experimentation, and is not necessarily optimal.

For each trimmer u/t, we submit $c_0 u^e t^{-e}$ to the padding oracle. If the oracle succeeds, we can trim the bounds of M_0.

A large denominator t allows for a more efficient trimming. The trimming process can be thus optimised by taking successful trimming fractions $u_1/t_1, \ldots, u_n/t_n$, computing the lowest common multiple t' of $t_1, \ldots, t_n$, using this value as a denominator and then searching for the highest and lowest numerators u_h, u_l that imply a valid padding, giving $2B \cdot t'/u_l \leq m < 3B \cdot t'/u_h$.

Skipping Holes In the original algorithm step 2a, the search for the first s_1 starts at the value $\lceil n/3B \rceil$. However, note that to be conforming we require in fact that $m \cdot s \geq n + 2B$. Since $3B - 1 \geq m$ we get $(3B-1)s \geq n+2B$. So we can start with $s = \lceil (n + 2B)/(3B - 1) \rceil$. On its own this does not save us much: about 8000 queries depending on the exact value of the modulus. However, when we have already applied the trimming rule above to reduce the upper bound on M_0 to some b, this translates immediately into a better start bound for s_1 of $(n + 2B)/b$.

Observe that in general for a successful s we must have $2B \leq ms - jn < 3B$ for some natural number j. Given that we have trimmed the first interval M_0 to the range $[a, b]$, this gives us a series of bounds

$$\frac{2B + jn}{b} \leq s < \frac{3B + jn}{a}$$

Observe further that when

$$\frac{3B + jn}{a} < \frac{2B + (j+1)n}{b}$$

we have a 'hole' of values where a suitable s cannot possibly be. When used in combination with the trimming rule, we found that we frequently obtain a list of many such holes. We use this list to skip out the holes during the search for the s_1. Note that this is similar to the reasoning used to calculate s values in step 2c, except that here we are concerned with finding the smallest possible s_1 in order to have the fewest possible intervals remaining when searching for s_2. As we show in the results below, the combination of the trimming and hole skipping techniques is highly effective, in particular against more permissive oracles than a strict PKCS padding oracle.

2.3 Existing Optimisations

In addition to our original modifications, we also implemented changes proposed by Klima, Pokorny and Rosa (KPR) [29]. These are mainly aimed at improving performance in step 2b, because they were concerned with attacking a weaker oracle where most time was spent in step 2b (see below). They are therefore naturally complementary to our optimisation of step 2a.

Parallel thread method The parallel thread method consists of omitting step 2b in the case where there are several intervals in M_{i-1}, and instead forking a separate thread for each interval and using the method of step 2c to search for s_i. As soon as one thread finds a hit, all threads are halted and the new intervals are calculated. If there is still more than one interval remaining, new threads are launched. In practice, since access to the oracle may not be parallelisable, the actions of each thread can be executed stepwise. This heuristic is quite powerful in practice, as we will see below.

Tighter bounds and Beta Method KPR were concerned with attacking the weaker 'bad version' oracle found in implementations of SSL patched against the original vulnerability. This meant that when the oracle succeeds, they could be sure of the length of the unpadded plaintext, since it must be the right length for the SSL 'pre-master secret'. This allowed them to tighten the $2B$ and $3B-1$ bounds. We also implemented this optimisation where possible, since it has no significant cost, but its effects are not significant. We implemented a further proposal of KPR, the so-called 'Beta Method' that we do not have space to describe here (see appendix A), but again found that it caused little improvement in practice.

2.4 Stronger and Weaker Oracles

In order to capture behaviour found in real devices (see section 3), we define stronger and weaker Bleichenbacher oracles, i.e. oracles which return true for a greater or smaller proportion of values x such that $2B \leq x < 3B$. We characterise them by three Booleans specifying the tests they apply or skip on the decrypted plaintext. The first Boolean corresponds to the test for a 0 somewhere after the first ten bytes. The second Boolean corresponds to the check for 0s in the non-zero padding. The third Boolean corresponds to a check of the plaintext length

against some specific value (e.g. 16 bytes for an encrypted AES-128 key). More precisely, we say an oracle is FFF if it returns true only on correctly padded plaintexts of a specific fixed length, like the the KPR 'bad version' oracle found in some old versions of SSL. An oracle is FFT if it returns true on a correctly padded plaintext of any length. This is the standard PKCS oracle used by Bleichenbacher. An oracle is FTT if it returns true on a correctly padded plaintext of any length and additionally on an otherwise correctly padded plaintext containing a zero in the eight byte padding. An oracle is TFT if if returns true on a correctly padded plaintext of any length and on plaintexts containing no 0s after the first byte. The most permissive oracle, TTT, returns true on any plaintext starting with `0x00, 0x02`. We will see in the next section how all these oracles arise in practice.

In Table 1, we show performance of the standard Bleichenbacher algorithm on these oracles, apart from FFF for which it is far too slow to obtain meaningful statistics. Attacking the strongest oracles TTT and TFT is substantially easier than the standard oracle. We can explain this by observing that for the original oracle, on a 1024 bit block, the probability $Pr(P)$ of a random ciphertext decrypting to give a conforming block is equal to the probability that the first two blocks are `0x00, 0x02`, the next 8 bytes are non-zero, and there is a zero somewhere after that. We let $Pr(A)$ be the probability that the first two bytes are `0x00, 0x02`, i.e $Pr(A) \approx 2^{-16}$. We identify $Pr(P|A)$, the probability of a ciphertext giving a valid plaintext provided the first two bytes are `0x00, 0x02`, i.e

$$\left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{118}\right) \approx 0.358$$

$Pr(P)$ is therefore $0.358 \cdot 2^{-16}$. Bleichenbacher estimates that, if no blinding phase is required, the attack on a 128 byte plaintext will take

$$2/Pr(P) + 16 \cdot 128/Pr(P|A)$$

oracle calls. So we have

$$(2 \cdot 2^{16} + 16 \cdot 128)/Pr(P|A) = 371843$$

In the case of, say, the TTT oracle, $Pr(P|A)$ is 1, since any block starting `0x00, 0x02` will be accepted. Hence we have

$$2^{17} + 16 \cdot 128 = 133120$$

oracle queries. This is higher than what we were able to achieve in practice in both cases, but the discrepancy is not surprising since the analysis Bleichenbacher uses is a heuristic approximation of the upper bound rather than the mean. However, it gives an explanation of why the powerful oracle gives such a big improvement in run times: improvements in the oracle to $Pr(P|A)$ make a multiplicative difference to the run time. Additionally, the expected number of intervals at the end of step 2a is $\lceil s_1 \cdot B/n \rceil$ [2, p. 7], so if s_1 is less than 2^{16}, the expected number of intervals is one. For the FFT oracle, the expected value of s_1 (calculated as $1/2 \cdot 1/Pr(P)$) is about 91 500, between 2^{16} and 2^{17}, whereas for TTT it is 2^{15}. That means that in the TTT case we can often jump step 2b and go straight to step 2c, giving a total of

$$2^{15} + 16 \cdot 128 = 34816$$

i.e. the TTT oracle is about 10 times more powerful than the FFT oracle, which is fairly close to what we see in practice (our mean for FFT is about 5.5 times that for TTT).

In comparison, if the modulus is 2048 bit long, then $Pr(P|A) \approx 0.599$. Because the modulus is longer, the probability that `0x00` appears after the 8 non-zero bytes is higher than in the 1024 bit case. Furthermore, following the same argument as above, we obtain that the attack on a 2048 bit plaintext will take about 335 065 calls to the FFT oracle, fewer than in the 1024 bit case. Note however that RSA private key operations slow down by roughly a factor of four when key length is doubled.

2.5 Performance of the Modified Algorithm

Referring again to Table 1, we give a summary of our experiments with our modified algorithm. As well as mean and median, we give the number of trimming fractions tried and the average number of oracle calls saved by the hole skipping modification we presented in section 2.2. Observe that as the oracles become stronger, the contribution of the KPR 'parallel threads' method becomes less significant and our hole skipping technique more significant. This is to be expected, since as discussed above, for the stronger oracles, fewer runs need to use step 2b. Similarly, when trimming the first interval M_0, we find that more fractions can be used because of the more permissive oracle, hence we find more holes to skip. For the most restrictive oracle, FFF, the addition of our trimming method slightly improves on the results of KPR (which were 20 835 297 mean and 13 331 256

Oracle	Original algorithm		Modified algorithm			
	Mean	Median	Mean	Median	Trimmers	Mean skipped
FFF	-	-	18 040 221	12 525 835	50 000	7 321
FFT	215 982	163 183	49 001	14 501	1 500	65 944
FTT	159 334	111 984	39 649	11 276	2 000	61 552
TFT	39 536	24 926	10 295	4 014	600	20 192
TTT	38 625	22 641	9 374	3 768	500	18 467

Table 1. Performance of the original and modified algorithms.

median). Note also that the trimming technique contributes more than just the oracle calls saved by the hole skipping, it also slightly improves performance on all subsequent stages of the algorithm. We know this because we can compare performance using only the parallel threads optimisation, where we obtain a mean of 113 667 and a median of 78 674 (on the FFT oracle). In Figure 1, we give the density distribution for 1000 runs of the original algorithm and our optimised algorithm on the classical FFT oracle, with medians marked. Notice the change in shape: we have a much thinner tail.

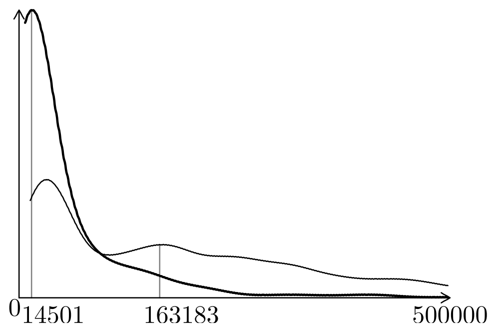

Fig. 1. Graph comparing distribution of oracle calls for original (lower peak, thinner line) and optimised version of the algorithm on the FFT oracle. Median is marked for each.

2.6 Vaudenay's Attack

Vaudenay's attack on CBC mode symmetric-key encryption [30] is somewhat simpler and highly efficient. Recall first the operation of CBC mode [9]: given some block cipher with encryption, decryption functions $E(.), D(.)$ and a fixed block size of b bytes, suppose we want to encrypt a message P of length $l = j \cdot b$ for some integer j, i.e. $P = P_1, \ldots, P_j$. In CBC mode, we first choose a fresh *initialisation vector* IV. The first encrypted block is defined as $C_1 = E(IV \oplus P_1)$, and subsequent blocks as $C_i = E(C_{i-1} \oplus P_i)$. The need for padding arises because l is not always a multiple of b. Suppose $l = j \cdot b + r$. Then we need to encrypt the last r bytes of the message in a b bytes block in such a way that on decryption, we can recognise that only the first r bytes are to be considered part of the plaintext. One way to do this is the so-called RC5 padding, also known as PKCS padding and described in RFC 5652 [14]. The r bytes are encoded into the leftmost bytes of the final block, and then the final $b - r$ bytes are filled with the value $b - r$. Under this padding scheme, if the plaintext length should happen to be an exact multiple of the block size, then we add a whole block of padding bytes b.

To effect Vaudenay's attack, suppose that the attacker has some ciphertext $C_1, \ldots, C_n$ and access to an oracle that returns true just when a ciphertext decrypts with valid padding. To attack a given block C_i, we first prepend a random block $R = r_1, \ldots, r_b$. We then ask the padding oracle to decrypt $R \mid C_i$. If the padding is valid most probably the final byte is 1, hence the final byte p_m of the plaintext P_i satisfies $p_b = r_b \oplus 1$. If the padding is not accepted, we iterate over i setting $r'_b = r_b \oplus i$ and retrying the oracle until eventually it is accepted. There is a small chance that the final byte of an accepted block is not 1, but this is easily detected. Having discovered the last byte, it is easy to extend the attack to obtain p_{b-1} by tweaking r_{b-1}, and so on for the whole block. Given this 'block decryption oracle' we can then apply it to all the blocks of the message. Overall, the attack requires $O(nb)$ steps, and hence is highly efficient.

Since the original attack appeared, many variations have been found on other padding schemes and block cipher modes [1, 6, 17, 20, 23, 25]. Bond and French recently showed that the attack could be applied to the `C_UnwrapKey` command as implemented on a hardware security module (HSM) [3]. We will show in the next section that many cryptographic devices are indeed vulnerable to variants of the attack.

Device	PKCS#11 version	PKCS#1 v1.5 Attack		CBC-PAD Attack	
		Token	Session	Token	Session
Aladdin eTokenPro	2.01	✓	✓	✓	✓
Feitian ePass 2000	2.11	×	×	N/A	N/A
Feitian ePass 3003	2.20	×	×	N/A	N/A
Gemalto Cyberflex	2.01	✓	N/A	N/A	N/A
RSA Securid 800	2.20	✓	N/A	N/A	N/A
Safenet Ikey 2032	2.01	✓	✓	N/A	N/A
SATA DKey	2.11	×	×	×	×
Siemens CardOS	2.11	✓	✓	N/A	N/A

Table 2. Attack Results on Tokens

3 Attacking Real Devices

We applied the optimised versions of the attacks of Bleichenbacher and Vaudenay presented in section 2 to the unwrap functionality of PKCS#11 devices. RSA PKCS#11, which describes the 'Cryptoki' API for cryptographic hardware, was first published in 1995 (v1.0). The latest official version is v2.20 (2004) which runs to just under 400 pages [28]. Adoption of the standard is almost ubiquitous in commercial cryptographic tokens and smartcards, even if other additional interfaces are frequently offered. In a PKCS#11-based API, applications initiate a *session* with the cryptographic token, by supplying a PIN. Once a session is initiated, the application may access the *objects* stored on the token, such as keys and certificates. Objects are referenced in the API via *handles*, which can be thought of as pointers to or names for the objects. In general, the value of the handle, e.g. for a secret key, does not reveal any information about the actual value of the key. Objects have *attributes*, which may be bitstrings e.g. the value of a key, or Boolean flags signalling properties of the object, e.g. whether the key may be used for encryption (`CKA_ENCRYPT`[6]), or for encrypting other keys, for signing, verification, and other uses. New objects can be created by calling a key generation command, or by *unwrapping* an encrypted key packet using the `C_UnwrapKey` command, which takes a handle, a ciphertext and a *template* as input. A template is a partial description of the key to be imported, giving notably its length. The device attempts to decrypt the ciphertext using the key referred to by the handle. If it succeeds, it creates a new key on the device using the extracted plaintext and the template, and returns a new handle.

Observe that a padding check immediately following the decryption could give rise to an oracle that may be used to determine the value of the newly stored key. To test for such an oracle on a device, we create a key with the `CKA_UNWRAP` attribute set to allow the `C_UnwrapKey` operation, create encrypted key packets with deliberately placed padding errors, call the function on these ciphertexts and observe the return codes. For the case of asymmetric key unwrapping, constructing test ciphertexts is easy since the public key of the pair is always obtainable via a query to the PKCS#11 interface. For symmetric key unwrapping, it is not quite so trivial since the device may create unwrapping keys marked with the Boolean key attribute `CKA_SENSITIVE` which prevents them from being read via the PKCS#11 interface. In this case there are various tricks we can use: we can try to set the attribute `CKA_ENCRYPT` and then use the PKCS#11 function `C_Encrypt` to construct the test packets if a suitable mode is available, or if the device does not allows this, we can explicitly try to create a key with `CKA_SENSITIVE` set to false, assuming the same unwrap algorithm will be used as for sensitive keys. In the event, we were always able to find some way to do this with the devices under test.

[6] Throughout the paper we will refer to commands, attributes, return codes and mechanisms by their names as defined in the PKCS#11 standard, so `C_` prefixes a (cryptoki) command, `CKA_` prefixes a cryptoki attribute, `CKR_` prefixes a cryptoki return code and `CKM_` prefixes a cryptoki mechanism.

3.1 Smartcards and Security Tokens

In Table 2 we give results from implementing the attacks on all the commercially available smartcards and USB tokens we were able to obtain that offer a PKCS#11 interface and support the unwrap operation. A tick means not only that we were able to construct a padding oracle, but that we were actually able to execute the attack and extract the correct encrypted key. A cross notes that the attack fails. We explain these failures below. Not applicable (N/A) means that the token did not support the cryptographic mechanisms and/or unwrap modes required for this attack. Note that relatively few devices support unwrap under symmetric key algorithms. We tested the attacks using both token keys and session keys for the unwrapping. The exact semantics of the difference between these key types is not completely clear from the standard: there is an attribute `CKA_TOKEN` which when set to true indicates a token key and when false indicates a session key. Session keys are destroyed when the session is ended, whereas token keys persist. However, we have noticed that devices often enforce very different policies for token keys and session keys, so it seemed pertinent to test both types.

In Table 3 we give the class of padding oracle found in each device in the PKCS#1 v1.5 case. To obtain this table we construct padded plaintexts with a single padding error and observed the return code from the token (the exact return codes are in the appendix, Table 4). Note that we give separate entries for token and session keys in this table only when there is a difference in the device's behaviour in the two cases. We report median attack time, computed from the results of table 1 and from a measure of the unwrap rate of the hardware. Notice how the tenfold improvement in median attack time of our modified algorithm makes attacks even against FFT oracles on slow devices quite practical. Unwrap calls using session keys are often many times faster than token keys though it is not clear why, unless perhaps these devices are carrying out session key operations in the driver software rather than on the card.

We will briefly discuss each line of Table 2 in turn. The **Aladdin eToken Pro** supports both unwrapping modes required, though the `CBC_PAD` unwrap mode does not conform to the standard: a block containing a final byte of `0x00` is accepted. According to the standard, if the final byte of the plaintext is zero and it falls at the end of a block, then an entire block of padding should be added (see section 2). This causes a small problem for the attack since it gives us an extra possibility for the last byte, but we easily adapted the attack to take account of this. The PKCS#1 v1.5 padding implementation ignores zeros in the first 8 bytes of the padding and gives a separate error when the length of the extracted key does not match the requested one (`CKR_TEMPLATE_INCONSISTENT`). Based on this we can build an FTT oracle. The **Feitian** tokens do not support `CBC_PAD` modes. They also do not implement PKCS#1 v1.5 padding correctly as shown in Table 4: in our tests, any block with `0x02` in the second byte was accepted, except for very large values (e.g. for one key, anything between `0x00` and `0xE2` in the first byte was accepted). The result is that the attack does not succeed. The **Gemalto Cyberflex** smartcard does not allow unwrapping under symmetric keys. However, it seems to implement standard PKCS#1 v1.5 padding correctly, and the Bleichenbacher attack succeeds (FFT oracle, since the length is ignored). The **RSA SecurID** device does not support unwrapping using symmetric keys, hence the Vaudenay attack is not possible. However, the Bleichenbacher attack works perfectly. In fact, the RSA token implements a perfect TTT oracle. The device also supports OAEP, but not in a way that prevents the attack (see next paragraph). The **Safenet ikey2032** implements an asymmetric key unwrapping. The padding oracle derived is more accepting than the Bleichenbacher oracle since the 0s in the first 8 bytes of the padding string are ignored (FTT oracle). The **SATA DKey** does not implement standard padding checks. In `CBC_PAD` mode, only the last byte is checked: it seems that as long as the last byte n is less than the number of bytes in a block, the padding is accepted and the final n bytes discarded. This means we cannot use the attack to recover the whole key, just the final byte. In PKCS#1 v1.5 mode, many incorrectly padded blocks were accepted, and we were unable to deduce the rationale. For example, any block with the *first* byte equal to `0x02` is accepted. The wide range of accepted blocks prevents the attack. The **Siemens CardOS** supports only unwrapping under asymmetric keys. The Bleichenbacher attack works perfectly: with token keys the oracle is TTT, while with session keys it is FFT.

Attacking OAEP Mode Unwrapping A solution to the Bleichenbacher attack is to use OAEP mode encryption, which was first added to PKCS#1 in v2.0 (1998) and is recommended for all new applications since v2.1 (2002). RSA OAEP was included as a mechanism in PKCS#11 in version 2.10 (1999). However, out of the tokens tested (all of which are currently available products), only one, the RSA SecureID, supports OAEP encryption. The standard PKCS#1 v2.1 notes that it is dangerous to allow two mechanisms to be enabled on the same key [27, p. 14], since "an opponent might be able to exploit a weakness in the implementation of RSAES-PKCS1-v1_5 to recover messages encrypted with either scheme.". An examination of the developer's manual for the RSA SecurID reveals that for private keys generated by the token, the relevant attribute "`CKA_ALLOWED_MECHANISMS`

Device	Token		Session	
	Oracle	Time	Oracle	Time
Aladdin eTokenPro	FTT	21m	FTT	17m
Gemalto Cyberflex	FFT	92m	N/A	N/A
RSA Securid 800	TTT	13m	N/A	N/A
Safenet Ikey 2032	FTT	88m	FTT	17m
Siemens CardOS	TTT	21m	FFT	89s

Table 3. Oracle Details and Median Attack Times

is always set to the following mechanism list : `CKM_RSA_PKCS`, `CKM_RSA_PKCS_OAEP`, and `CKM_RSA_X_509.`". We created a key wrapped under OAEP and then performed Bleichenbacher's attack on it using a PKCS#1 v1.5 unwrap oracle. The attack is only slightly complicated by the fact that the initial encrypted block does not yield a valid block when decrypted, requiring us to use the 'blinding phase' where many ciphertexts are derived form the original to obtain one that passes the padding oracle. In our tests this added only a few hundred seconds to the attack.

3.2 HSMs

Hardware Security Modules are widely used in banking and similar sectors where a large amount of cryptographic processing has to be done securely at high speed (verifying PIN numbers, signing transactions, etc.). A typical HSM retails for around 20 000 Euros hence is unfortunately too expensive for our laboratory budget. HSMs process RSA operations at considerable speed: over 1000 decryptions per second for 1024 bit keys. Even in the case of the FFF oracle, which requires 12 000 000 queries, this would result in a median attack time of 12 000 seconds, or just over three hours.

We hope to be able to give details of HSM testing soon.

3.3 Estonian ID Card

Estonia's Citizenship and Migration Board completed the issuing of more than 1 million national electronic ID (eID) cards in 2006 [19]. The eID is the primary national identification document in Estonia and it is mandatory for all Estonian citizens and alien residents 15 years and older to have one [10]. The card contains two RSA key pairs [15]. One key pair is intended to be mainly used for authentication (e.g., for mutual authentication with TLS/SSL) but can also be used for encrypting and signing email (e.g., with S/MIME). The other key pair is attributed only to be used for digital signatures. Only this latter key pair can be used for legally binding digital signatures [19]. Since January 1, 2011, the eID cards contain 2048 bit RSA keys, therefore these cards comply with NIST's recommendation [21]. However, cards issued before January 1, 2011 continue to use 1024 bit keys.

Attack Vector Unlike the cryptographic devices discussed above, the Estonian eID card does not allow the import of keys, so our attack here does not rely on the unwrap operation. Instead we consider attacks using the padding oracle provided by the decryption function of the DigiDoc software, part of the official ID software package developed by the Estonian Certification Center, Estonia's only CA [11]. We note that the attack succeeds with any application that returns whether decryption with the eID card succeeds. Our experiments were conducted using the Java library of DigiDoc, called *JDigiDoc*. DigiDoc encrypts data using a hybrid encryption scheme, where a 128-bit AES key is encrypted under a public key. First we tested the Estonian ID card's decryption function using raw PKCS#11 calls and confirmed that it checks padding correctly. We then observed that with the default configuration, when attempting to decrypt, e.g., an encrypted email, JDigiDoc writes a log file of debug information that includes the padding errors for the 128-bit AES key that is encrypted under the public key. This behavior has been observed with JDigiDoc version 2.3.19, and the latest version (3.6.0.157) does not seem to change it. Any application built on JDigiDoc, that reveals whether decryption succeeds, e.g., by leaking the contents of the log file, provides an attacker with a suitable padding oracle. The information in JDigiDoc's log file gives an attacker access to essentially an FFT oracle but with additional length information. The length information allows us to adjust the $2B$ and $3B - 1$ bounds used in the attack, though in our experiments this made little difference.

In tests, the Estonian ID card, using 2048 bit keys, was able to perform 100 decryptions in 340 seconds. This means that for our optimised attack, where 28 300 decryptions are required, we would need about 96 200

seconds, or about 27 hours to decrypt an arbitrary valid ciphertext. For ID cards using 1024 bit keys, each decryption should be four times faster, while 49 000 decryptions are required; therefore we estimate a time of about 41 700 seconds, or about 11 hours and 30 minutes to decrypt an arbitrary valid ciphertext. To forge a signature, we require, due to the extra blinding step, a mean of 109 000 oracle calls and a median of 69 000 oracle calls to get a valid signature on an arbitrary message, giving an expected time of 103 hours on a 2048 bit Estonian eID. On a card using 1024 bit keys, we require a mean of 203 000 calls and a median of 126 000 calls; therefore expect to sign an arbitrary message in around 48 hours.

4 Countermeasures

A general countermeasure to the Bleichenbacher and Vaudenay attacks has been well known for years: use authenticated encryption. There are no such modes for symmetric key encryption in the current version of PKCS#11, but version 2.30, which is still at the draft stage, includes GCM and CCM (mechanisms `CKM_AES_GCM` and `CKM_AES_CCM`). While these modes have their critics [26], they do in theory provide secure authenticated encryption and hence could form the basis of secure symmetric key unwrap mechanisms. Unfortunately, in the current draft (v7), they are given only as modes for `C_Encrypt`. Adoption of these modes for `C_UnwrapKey` would provide a great opportunity to give the option of specifying authenticated data along with the encrypted key to allow secure transfer of attributes between devices. This would greatly enhance the flexibility of secure configurations of PKCS#11. To prevent the Bleichenbacher attack one must simply switch to OAEP, which is already in the standard. PKCS#11 should follow PKCS#1's long-held position of recommending OAEP exclusively for all new applications. Care must also be taken to remind developers not to allow the two modes to be used on the same key, as is the case in RSA's own SecureID device. In fact, the minutes of the 2003 PKCS workshop suggest that there was a consensus to include the single mechanism recommendation in version 2.20 [24], but it does not appear in the final draft. Note that care must be taken when implementing OAEP as otherwise there may also be a padding oracle attack which is even more efficient than our modified Bleichenbacher attack [18], though we are yet to find such an oracle on a PKCS#11 device.

If unauthenticated unwrap modes need to be maintained for backwards compatibility reasons, there are various options available. For the CBC case, Black and Urtubia note that the 10^* padding, where the plaintext is followed by a single 1 bit and then only 0 bits until the end of the block, leaks no information from failed padding checks while still allowing length of the plaintext to be determined unambiguously [1]. Paterson and Watson suggest a refinement that additionally preserves a notion of indistinguishability, by ensuring that no padded blocks are invalid [22]. They also give appropriate security proofs for the two schemes. If PKCS#1 v1.5 needs to be maintained, we have seen that an implementation of the padding check that rejects anything other than a conforming plaintext containing a key of the correct length with a single error code gives the weakest possible (FFF) oracle. This may be enough for some applications, but one is well advised to remember the maxim that attacks only get better, never worse. An alternative approach would be to adopt 'SSL style' countermeasures, proceeding to import a randomly generated key in the case where a block contains invalid padding. However, this may not fix the hole: if an attacker is able to replay the same block and detect that two different keys have been imported, he knows there is a padding error. One could also decide to ignore padding errors completely and always import just the number of bytes corresponding to the size of the key required, but this looks dangerous: if the same block can be passed off as several different kinds of key, this might open the possibility of attacking weaker algorithms to obtain keys for stronger ones. Thus it seems clear that authenticated encryption is by far the superior solution.

We detail manufacturer responses in Appendix C. There is a broad spectrum: while some manufacturers offer mitigations and state a clear need to get authenticated encryption into the standard and adopted as soon as possible, others see their responsibility as ending as soon as they conform to the PKCS#11 standard, however vulnerable it might be.

5 Conclusions

We have demonstrated a modified version of the Bleichenbacher RSA PKCS#1 v1.5 attack that allows the 'million message attack' to be carried out in a few tens of thousands of messages in many cases. We have implemented and tested this and the Vaudenay CBC attack on a variety of contemporary cryptographic hardware, enabling us to determine the value of encrypted keys under import. We have shown that the way the `C_UnwrapKey` command from the PKCS#11 standard is implemented on many devices gives rise to an especially powerful error oracle that further reduces the complexity of the Bleichenbacher attack. In the worst case, we

found devices for which our algorithm requires a median of only 3 800 oracle calls to determine the value of the imported key. Vulnerable devices include eID cards, smartcards and USB tokens.

Related work Most previous work on the security of PKCS#11 interfaces considers only attacks at the logical level, i.e. in the symbolic or Dolev-Yao model of cryptography [7, 8, 12, 13]. Clulow's original assessment did discuss some cryptographic details, in particular a possible vulnerability if 'raw' unpadded RSA encryption is used to encrypt a symmetric key with a low public exponent (i.e. 3 [5, §3.3]). If e.g. a 128-bit symmetric key is right justified and left padded with 0s in a 1024 bit block and then raised to the power of 3, it will result in a value not greater than the modulus, hence the attacker only needs to calculate the cube root to crack the key. The tokens we tested all fix their public exponent at 65 537. Clulow notes that the private key unwrap operation includes a padding check [5, §3], but does not discuss the possibility of padding oracle attacks. Bond and French discussed a symmetric key unwrap oracle attack in a presentation at the 2010 API Analysis Workshop [3], though they did not reveal details of the hardware tested or discuss an asymmetric key version. Recently, Jager et al. showed that the Bleichenbacher attack can be used against XML encryption in many implementations [16]. Our optimisations could be applied directly to their setting to make their attacks more efficient.

While some theoreticians find the lack of a security proof sufficient grounds for rejecting a scheme, some practitioners find the absence of practical attacks sufficient grounds for continuing to use it. We hope that the new results with our modified algorithm will prompt editors to reconsider the inclusion of PKCS#1 v1.5 in contemporary standards such as PKCS#11 and XML encryption.

References

1. John Black and Hector Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In Dan Boneh, editor, *USENIX Security Symposium*, pages 327–338. USENIX, 2002.
2. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard. In *Advances in Cryptology: Proceedings of CRYPTO '98*, volume 1462 of *LNCS*, pages 1–12, 1998.
3. Mike Bond and George French. Hidden semantics: why? how? and what to do? Presentation at Fourth Analysis of Security APIs workshop (ASA-4), July 2010.
4. Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, Illinois, USA, October 2010. ACM Press.
5. J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 411–425, 2003.
6. Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of ipsec in mac-then-encrypt configurations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 493–504. ACM, 2010.
7. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.
8. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 2009. To appear.
9. M. Dworkin. Recommendation for block cipher modes of operation: Modes and techniques. NIST Special Publication 800-38A, December 2001.
10. Estonian Certification Center. The estonian ID card and digital signature concept, principles and solutions. `http://www.id.ee/public/The_Estonian_ID_Card_and_Digital_Signature_Concept.pdf`, March 2003.
11. Estonian Informatics Center. Estonian ID-software. `https://installer.id.ee/?lang=eng`.
12. Sibylle Fröschle and Graham Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In Pierpaolo Degano and Luca Viganò, editors, *Revised Selected Papers of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, August 2009. Springer.
13. Sibylle B. Fröschle and Nils Sommer. Reasoning with past to prove PKCS#11 keys secure. In Pierpaolo Degano, Sandro Etalle, and Joshua D. Guttman, editors, *Formal Aspects in Security and Trust*, volume 6561 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2010.
14. R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Standard), September 2009.
15. ID Süsteemide AS. EstEID specification v2.01. `http://www.id.ee/public/EstEID_Spetsifikatsioon_v2.01.pdf`.
16. T. Jager, S. Schinzel, and J. Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML encryption. In *17th European Symposium on Research in Computer Security (ESORICS)*, 2012. To appear.
17. T. Jager and J. Somorovsky. How to break XML encryption. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 413–422, 2011.

18. James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer Berlin / Heidelberg, 2001.
19. Tarvi Martens. eID interoperability for PEGS, national profile estonia, European Commission's IDABC programme. `http://ec.europa.eu/idabc/en/document/6485/5938`, November 2007.
20. Chris J. Mitchell. Error oracle attacks on CBC mode: Is there a future for CBC mode encryption? In J. et al. Zhou, editor, *ISC 2005*, number 3650 in LNCS, pages 244–258, 2005.
21. National Institute of Standards and Technology. NIST special publication 800-57, recommendation for key management. `http://csrc.nist.gov/publications/PubsSPs.html`, March 2007.
22. Kenneth G. Paterson and Gaven J. Watson. Immunising cbc mode against padding oracle attacks: A formal security treatment. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 340–357. Springer, 2008.
23. K.G. Paterson and A. Yau. Padding oracle attacks on the ISO CBC mode encryption standard. In T. Okamoto, editor, *RSA '04 Cryptography Track*, number 2964 in LNCS, pages 305–323. Springer, 2004.
24. Minutes from the April, 2003 PKCS workshop. Available at `ftp://ftp.rsa.com/pub/pkcs/03workshop/minutes.txt`, 2003.
25. Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
26. Phillip Rogaway. Evaluation of some blockcipher modes of operation. `http://www.cs.ucdavis.edu/~rogaway`, February 2011. Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan.
27. RSA Security Inc., v2.1. *PKCS #1: RSA Cryptography Standard*, June 2002.
28. RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.
29. T. Rosa V. Klima, O. Pokorny. Attacking RSA-based sessions in SSL/TLS. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 426 – 440. Springer-Verlag, 2003.
30. Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

A Modified Bleichenbacher Algorithm

We present the algorithm of the optimised Bleichenbacher attack. It incorporates existing and new optimisations as presented in section 2.2. Notation is as before.

Step 1 - Initialization

Step 1.a - Blinding For an integer c, choose different random integers s_0 and check whether $c \cdot (s_0)^e \ mod \ n$ is PKCS conforming, by accessing the padding oracle. (If $c \ mod \ n$ is conforming then choose $s_0 \leftarrow 1$ instead.) For the first successful value s_0, set $c_0 \leftarrow c \cdot (s_0)^e \ mod \ n$, $M_0 \leftarrow \{[2B, 3B-1]\}$, $i \leftarrow 1$.

Step 1.b - Trimming M_0 Generate pairs of coprime integers and, for each pair (u, t), check whether $c_0 u^e t^{-e} \ mod \ n$ is PKCS conforming. For successful pairs $(u_1, t_1), (u_2, t_2), \ldots, (u_q, t_q)$, compute the lowest common multiple t' of $t_1, t_2, \ldots, t_q$, search for the smallest integer $u_{\min}$ and the largest integer $u_{\max}$ such that $c_0 u_{\min}^e t'^{-e} \ mod \ n$ and $c_0 u_{\max}^e t'^{-e} \ mod \ n$ are PKCS conforming. Set

$$\begin{aligned} a &\leftarrow 2B \cdot t'/u_{\min} \\ b &\leftarrow (3B-1) \cdot t'/u_{\max} \\ M_0 &\leftarrow \{[a, b]\}. \end{aligned}$$

Step 2 - Searching for PKCS conforming message

Step 2.a - Starting the search while Skipping Holes If $i = 1$, then search for the smallest positive integer $s_1 \geq \lceil (n + 2B)/b \rceil$ such that $c_0 \cdot s_1^e \ mod \ n$ is PKCS conforming. While searching for s_1, skip all values s' such that

$$(3B + jn)/a \leq s' < (2B + (j+1)n)/b$$

and do not access the padding oracle to check whether $c_0 \cdot s'^e \ mod \ n$ is PKCS conforming.

Step 2.b - Searching with more than one interval left If $i > 1$ and $|M_{i-1}| > 1$, then

Step 2.b.i - Parallel Threads Method If $|M_{i-1}| \leq P_{\max}$[7], then for each interval $I_j \in M_{i-1}$, start its own thread T_j following Step 2.c, for $j = 1, 2, \ldots, |M_{i-1}|$. The threads T_j take rounds making each one oracle call per round. If one of the threads finds a s_i such that $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming, then go to Step 3.

Step 2.b.ii - Beta Method [8] If $|M_{i-1}| > P_{\max}$, then search for the smallest integer $2 \leq \beta \leq \beta_{\max}$[9] such that for

$$s_i \leftarrow \beta s_{i-1} - (\beta - 1)s_0$$

$c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming. If failed to find s_i, go to Step 2.b.iii.

Step 2.b.iii - No optimisation If Step 2.b.ii failed, then search for the smallest integer $s_i > s_{i-1}$ such that $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming. If such a s_i is found, go to Step 3.

Step 2.c - Searching with one interval left If $i > 1$ and $|M_{i-1}| = 1$, i.e., $M_{i-1} = \{[a, b]\}$, then choose small integers r_i, s_i such that

$$r_i \geq 2\frac{bs_{i-1} - 2B}{n}$$
$$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

until $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming.

Step 3 - Narrowing the set of solutions After s_i is found, let

$$M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor\right) \right] \right\}$$

for all $[a, b] \in M_{i-1}$ and $\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$.

Step 4 - Computing Solution If $M_i = [a, a]$, then set $m \leftarrow a(s_0)^{-1} \ mod\ n$, and return m as solution of $m \equiv c^d \ mod\ n$. Otherwise, set $i \leftarrow i + 1$ and continue with Step 2.b or Step 2.c.

B Actual Padding Errors Reported by Smartcards and USB Tokens

Table 4 reports actual padding errors returned by the devices we tested.

Device	First byte not `0x00`	Second byte not `0x02`	`0x00` in first 8 bytes padding	No `0x00` from byte 3 to 128	Length incorrect
Aladdin eToken PRO	1	1	4	1	4
Feitian epass 2000	0	5	5	5	0
Feitian epass 3003	0	3	5	5	5
Gemalto Cyberflex	2	2	2	2	0
RSA SecureID 800	1	1	0	0	0
Safenet Ikey 2032	1	1	4	1	4
SATA Dkey (session)	1	0	5	5	1
SATA Dkey (token)	1	1	5	5	1
Siemens CardOS (session)	5	5	5	5	0
Siemens CardOS (token)	5	5	0	0	5

Table 4. Variations found on PKCS#1 v1.5 Padding Tests. Error 0 = `CKR_OK` (key is imported), Error 1 = `CKR_ENCYRYPTED_DATA_INVALID`, Error 2 = `CKR_WRAPPED_KEY_INVALID`, Error 3 = `CKR_DATA_LEN_RANGE`, Error 4= `CKR_TEMPLATE_INCONSISTENT`, Error 5 = `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_DEVICE_ERROR` or similar.

[7] In practice we take $P_{\max} = 40$.
[8] We did not use beta method for most experiments. (See section 2.5.)
[9] In practice we take $\beta_{\max} = 40$.

C Manufacturer Reaction

We have notified all manufacturers of our findings in May 2011 and we summarize their reactions so far.

SafeNet released a security bulletin confirming the vulnerability on eToken Pro, eToken Pro Smartcard, eToken NG-OTP, eToken NG-FLASH, iKey 2032 using Aladdin eToken PKI Client or SafeNet Authentication Client software. As a workaround they suggest to use SafeNet Authentication Client 8.0 or later to enable PKCS#1 v2.1 padding for RSA and to avoid wrapping symmetric keys using other symmetric keys. They plan enhancements in their products for enabling symmetric keys wrapping with other symmetric keys using GCM and CCM modes of operation (discussed in section 4). They also plan to add a key wrapping policy that enforces the usage of only GCM and CCM modes of operation for symmetric encryption, and PKCS#1 v2.1 padding for RSA encryption.

RSA recognises that an attacker can obtain the corresponding plaintext through a padding oracle attack against RSA SecureID faster than would be possible with standard Bleichenbacher attack. They however claim that "this attack is unnecessary since the prerequisites to the attack are already enough to call `C_UnwrapKey` and `C_GetAttributeValue` and receive the same plaintext". Instead, they regard these flaws as incomplete compliance with the standard and they are planning to fix this in version 3.5.4 of the middleware. It is not clear if the firmware on the device will be fixed. Our perspective is that (1) full compliance with the standard would only slow down the attacks and not prevent them; (2) the attacker could have indirect attacks to the unwrapping functionality without accessing other functionalities such as `C_GetAttributeValue` and without knowing the PIN, e.g. though a network protocol

Siemens has also recognised the flaws and we have been informally told that they have fixed the verification of the padding and added a check of the obtained plaintext with respect to the given key template in the most recent version. Additionally, CardOS is no longer a product of Siemens. Due to the merge of Atos Origin and Siemens IT Solutions, the new product owner is Atos.

Gemalto did not respond to our initial vulnerability report but at the time of writing were investigating the issue.

We filed a vulnerability report of our attack on the Estonian eID card to the Estonian Certification Center in November 2011. They showed concern about the vulnerability of the card we reported and informed CERT Estonia about the flaw. However, according to the Estonian Certification Center the authentication certificate is mainly used for authentication with SSL (in 95% of the cases), and our attack would be too slow to forge an SSL client response before a server timeout. At the time of our communication they had not decided on any countermeasures. The most recent release (v3.6.0.157) of digiDoc does not change the default output to the debug file.