Practical Cryptanalysis of ISO 9796-2 and EMV Signatures

Jean-Sébastien Coron[1], David Naccache[2], Mehdi Tibouchi[3], and Ralf-Philipp Weinmann[1]

[1] Université du Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg, Luxembourg
{jean-sebastien.coron, ralf-philipp.weinmann}@uni.lu
[2] École normale supérieure
Département d'informatique, Groupe de Cryptographie
45, rue d'Ulm, F-75230 Paris Cedex 05, France
david.naccache@ens.fr
[3] NTT Secure Platform Laboratories
3–9–11 Midori-cho, Musashino-shi, Tokyo 180–8585, Japan
tibouchi.mehdi@lab.ntt.co.jp

Abstract. At Crypto 1999, Coron, Naccache and Stern described an existential signature forgery against two popular RSA signature standards, ISO 9796-1 and ISO 9796-2. Following this attack ISO 9796-1 was withdrawn, and ISO 9796-2 was amended by increasing the message digest to at least 160 bits. In this paper we describe an attack against the amended version of ISO 9796-2, for all modulus sizes. Our new attack is based on Bernstein's algorithm for detecting smooth numbers, instead of trial division. In practice we were able to compute a forgery in only two days on a network of 19 servers. Our attack can also be extended to EMV signatures, an ISO 9796-2-compliant format with extra redundancy.
In response to this new attack, the ISO 9796-2 standard was amended again in late 2010.

Keywords: public-key cryptanalysis, RSA signatures, ISO 9796-2, EMV.

1 Introduction

RSA Signatures. RSA [50] is certainly the most popular public-key cryptosystem. A chosen-ciphertext attack against RSA textbook encryption was described by Desmedt and Odlyzko in [21]. As noted in [43], Desmedt and Odlyzko's attack also applies to RSA signatures:

$$\sigma = \mu(m)^d \mod N$$

where $\mu(m)$ is an encoding function and d the private exponent. Desmedt and Odlyzko's attack only applies if the encoding function $\mu(m)$ is much smaller than

N. In which case, one obtains an existential forgery under a chosen-message attack: the opponent can ask for signatures of any messages of his choosing before computing, by his own means, the signature of a (possibly meaningless) message which was never signed by the legitimate owner of d.

One can distinguish two classes of encoding functions $\mu(m)$:

1. *Ad-hoc encodings* are "handcrafted" to thwart certain classes of attacks. While still in use, *ad-hoc* encodings are currently being phased-out. PKCS #1 v1.5 [33], ISO 9796-1 [28] and ISO 9796-2 [29, 30] are typical *ad-hoc* encoding examples.
2. *Provably secure encodings* are designed to make cryptanalysis equivalent to inverting RSA (generally in the Random Oracle Model [2]). OAEP [3] (for encryption) and PSS [4] (for signature) are typical provably secure encoding examples.

For *ad-hoc* encodings, there is no guarantee that forging signatures is as hard as inverting RSA, and many such encodings were found to be weaker than the RSA problem. We refer the reader to [11, 15, 14, 32, 18, 25] for a few characteristic examples. It is thus a practitioner's rule of thumb to use provably secure encodings whenever possible. Nonetheless, *ad-hoc* encodings continue to populate hundreds of millions of commercial products (*e.g.* EMV cards) for a variety of practical reasons. A periodic re-evaluation of such encodings is hence necessary.

The ISO 9796-2 Standard. ISO 9796-2 is a specific encoding function $\mu(m)$ standardized by ISO in [29]. At Crypto 1999, Coron, Naccache and Stern described an attack against ISO 9796-2 [19]. Their attack is an adaptation of Desmedt and Odlyzko's cryptanalysis which could not be applied directly since in ISO 9796-2, the encoding $\mu(m)$ is almost as large as the modulus N.

ISO 9796-2 can be used with hash-functions of diverse digest-sizes k_h. Coron *et al.* estimated that attacking $k_h = 128$ and $k_h = 160$ would require (respectively) 2^{54} and 2^{61} operations. After Coron *et al.*'s publication ISO 9796-2 was amended and the official requirement (see [30]) became $k_h \geq 160$. It was shown in [16] that ISO 9796-2 can be proven secure in the random oracle model for $e = 2$ and if the digest size k_h is a least 2/3 the size of the modulus.

Our new attack. In this paper, we describe an improved attack against the amended version of ISO 9796-2, that is for $k_h = 160$. The new attack applies to EMV signatures as well. EMV is an ISO 9796-2-compliant format with extra redundancy. Our new attack is similar to Coron *et al.* forgery but using Bernstein's smoothness detection algorithm instead of trial division; we also use some algorithmic refinements: better message choice, large prime variant and optimized exhaustive search.

In practice we were able to compute forgery for ISO 9796-2 in only two days, using a few dozens of servers on the Amazon EC2 grid, for a total cost of US$800. The forgery was implemented for $e = 2$ but attacking odd exponents would not take significantly longer.[1] We estimate that under similar conditions an EMV signature forgery would cost US$45,000. Note that all costs are per modulus; after computing a first forgery for a given N, additional forgeries come at a negligible cost.

2 The ISO 9796-2 Standard

ISO 9796-2 is an encoding standard allowing partial or total message recovery [29, 30]. Here we consider only partial message recovery. As already mentioned, ISO 9796-2 can be used with hash-functions $H(m)$ of diverse digest-sizes k_h. For the sake of simplicity we assume that the hash size k_h, the size of m and the size of N (denoted k) are all multiples of 8; this is also the case in the EMV specifications. The ISO 9796-2 encoding function is then:

$$\mu(m) = \texttt{6A}_{16} \| m[1] \| H(m) \| \texttt{BC}_{16}$$

where the message $m = m[1]\|m[2]$ is split in two: $m[1]$ consists of the $k - k_h - 16$ leftmost bits of m and $m[2]$ represents all the remaining bits of m. The size of $\mu(m)$ is therefore always $k - 1$ bits.

The original version of the standard recommended $128 \leq k_h \leq 160$ for partial message recovery (see [29], §5, note 4). The new version of ISO 9796-2 [30] requires $k_h \geq 160$. The EMV specifications also use $k_h = 160$.

Rabin-Williams Signatures. Since our attack will be implemented for $e = 2$, we briefly recall Rabin-Williams signatures. Such signatures use an encoding function $\mu(m)$ such that $\mu(m) = 12 \bmod 16$ for all m. In contrast with RSA, it is required that $p = 3 \bmod 8$ and $q = 7 \bmod 8$. For $e = 2$ the private key is $d = (N - p - q + 5)/8$. To sign a message m, first compute the Jacobi symbol $\mathrm{J} = \left(\frac{\mu(m)}{N}\right)$. The signature of m is then $s = \min(\sigma, N - \sigma)$, where:

$$\sigma = \begin{cases} \mu(m)^d \bmod N & \text{if } \mathrm{J} = 1 \\ (\mu(m)/2)^d \bmod N & \text{otherwise} \end{cases}$$

To verify the signature σ compute $\omega = s^2 \bmod N$ and check that:

$$\mu(m) \overset{?}{=} \begin{cases} \omega & \text{if } \omega = 4 \mod 8 \\ 2 \cdot \omega & \text{if } \omega = 6 \mod 8 \\ N - \omega & \text{if } \omega = 1 \mod 8 \\ 2 \cdot (N - \omega) & \text{if } \omega = 7 \mod 8 \end{cases}$$

[1] The size of the public exponent affects the linear algebra step of the attack slightly: for $e = 3$ or $e = 65537$, say, that step would take a bit longer, both because some matrices involves would be a bit less sparse, and because available sparse linear algebra packages are particularly optimized for the binary case, which is used in the quadratic sieve (on modern CPUs, they use bit packing, bit fiddling and other tricks). Nevertheless, the impact would be minor, particularly because the linear algebra step takes negligible time compared to the search for smooth numbers, for which the value of e is irrelevant.

The following fact shows that the Rabin-Williams signature verification works [41]. In particular, the fact that $\left(\frac{2}{N}\right) = -1$ ensures that either $\mu(m)$ or $\mu(m)/2$ has a Jacobi symbol equal to 1.

Fact 1. *Let N be an RSA-modulus with $p = 3 \mod 8$ and $q = 7 \mod 8$. Then $\left(\frac{2}{N}\right) = -1$ and $\left(\frac{-1}{N}\right) = 1$. Let $d = (N - p - q + 5)/8$. Then for any integer x such that $\left(\frac{x}{N}\right) = 1$, we have that $x^{2d} = x \mod N$ if x is a square modulo N, and $x^{2d} = -x \mod N$ otherwise.*

3 Desmedt-Odlyzko's Attack

Desmedt and Odlyzko's attack is an existential forgery under a chosen-message attack, in which the forger asks for the signature of messages of his choice before computing the signature of a (possibly meaningless) message that was never signed by the legitimate owner of d. In the case of Rabin-Williams signatures it may even happen that the attacker factors N; *i.e.* a total break. The attack only applies if $\mu(m)$ is much smaller than N and works as follows:

1. Select a bound B and let $\mathfrak{P} = \{p_1, \ldots, p_\ell\}$ be the list of all primes less or equal to B.
2. Find at least $\tau \geq \ell + 1$ messages m_i such that each $\mu(m_i)$ is a product of primes in $\mathfrak{P}$.
3. Express one $\mu(m_j)$ as a multiplicative combination of the other $\mu(m_i)$, by solving a linear system given by the exponent vectors of the $\mu(m_i)$ with respect to the primes in $\mathfrak{P}$.
4. Ask for the signatures of the m_i for $i \neq j$ and forge the signature of m_j.

In the following we assume that e is prime; this includes $e = 2$. We let τ be the number of messages m_i obtained at step 2. We say that an integer is B-smooth if all its prime factors are less or equal to B. The integers $\mu(m_i)$ obtained at step 2 are therefore B-smooth and we can write for all messages m_i, $1 \leq i \leq \tau$:

$$\mu(m_i) = \prod_{j=1}^{\ell} p_j^{v_{i,j}} \tag{1}$$

To each $\mu(m_i)$ we associate the ℓ-dimensional vector of the exponents modulo e, that is $\boldsymbol{V_i} = (v_{i,1} \bmod e, \ldots, v_{i,\ell} \bmod e)$. Since e is prime, the set of all ℓ-dimensional vectors modulo e forms a linear space of dimension ℓ. Therefore, if $\tau \geq \ell + 1$, one can express one vector, say $\boldsymbol{V_\tau}$, as a linear combination of the others modulo e, using Gaussian elimination:

$$\boldsymbol{V_\tau} = \boldsymbol{\Gamma} \cdot e + \sum_{i=1}^{\tau-1} \beta_i \boldsymbol{V_i}$$

for some $\boldsymbol{\Gamma} = (\gamma_1, \ldots, \gamma_\ell) \in \mathbb{Z}^\ell$ and some $\beta_i \in \{0, \ldots, e-1\}$. This gives for all $1 \leq j \leq \ell$:

$$v_{\tau,j} = \gamma_j \cdot e + \sum_{i=1}^{\tau-1} \beta_i \cdot v_{i,j}$$

Then using (1), one obtains:

$$\mu(m_\tau) = \prod_{j=1}^{\ell} p_j^{v_{\tau,j}} = \prod_{j=1}^{\ell} p_j^{\gamma_j \cdot e + \sum\limits_{i=1}^{\tau-1} \beta_i \cdot v_{i,j}} = \left(\prod_{j=1}^{\ell} p_j^{\gamma_j}\right)^e \cdot \prod_{j=1}^{\ell} \prod_{i=1}^{\tau-1} p_j^{v_{i,j} \cdot \beta_i}$$

$$\mu(m_\tau) = \left(\prod_{j=1}^{\ell} p_j^{\gamma_j}\right)^e \cdot \prod_{i=1}^{\tau-1} \left(\prod_{j=1}^{\ell} p_j^{v_{i,j}}\right)^{\beta_i} = \left(\prod_{j=1}^{\ell} p_j^{\gamma_j}\right)^e \cdot \prod_{i=1}^{\tau-1} \mu(m_i)^{\beta_i}$$

That is:

$$\mu(m_\tau) = \delta^e \cdot \prod_{i=1}^{\tau-1} \mu(m_i)^{\beta_i}, \quad \text{where } \delta := \prod_{j=1}^{\ell} p_j^{\gamma_j} \tag{2}$$

Therefore, we see that $\mu(m_\tau)$ can be written as a multiplicative combination of the other $\mu(m_i)$. For RSA signatures, the attacker will ask for the signatures σ_i of $m_1, \ldots, m_{\tau-1}$ and forge the signature σ_τ of m_τ using the relation:

$$\sigma_\tau = \mu(m_\tau)^d = \delta \cdot \prod_{i=1}^{\tau-1} \left(\mu(m_i)^d\right)^{\beta_i} = \delta \cdot \prod_{i=1}^{\tau-1} \sigma_i^{\beta_i} \pmod{N}$$

Rabin-Williams Signatures. For Rabin-Williams signatures ($e = 2$), the attacker may even factor N. Let $\mathrm{J}(x)$ denote the Jacobi symbol of x with respect to N. We distinguish two cases. If $\mathrm{J}(\delta) = 1$, we have $\delta^{2d} = \pm\delta \mod N$, which gives from (2) the forgery equation:

$$\mu(m_\tau)^d = \pm\delta \cdot \prod_{i=1}^{\tau-1} \left(\mu(m_i)^d\right)^{\beta_i} \pmod{N}$$

If $\mathrm{J}(\delta) = -1$, then letting $u = \delta^{2d} \bmod N$ we obtain $u^2 = (\delta^2)^{2d} = \delta^2 \bmod N$, which implies $(u-\delta)(u+\delta) = 0 \bmod N$. Moreover since $\mathrm{J}(\delta) = -\mathrm{J}(u)$ we must have $\delta \neq \pm u \mod N$ and therefore $\gcd(u \pm \delta, N)$ will factor N. The attacker can therefore submit the τ messages for signature, recover $u = \delta^{2d} \bmod N$, factor N and subsequently sign any message.[2]

[2] In both cases we have assumed that the signature is always $\sigma = \mu(m)^d \bmod N$, whereas by definition a Rabin-Williams signature is $\sigma = (\mu(m)/2)^d \bmod N$ when $\mathrm{J}(\mu(m)) = -1$. A possible work-around consists in discarding such messages but it is also easy to adapt the attack to handle both cases.

Attack complexity. The complexity of the attack depends on the number of primes ℓ and on the probability that the integers $\mu(m_i)$ are p_ℓ-smooth, where p_ℓ is the ℓ-th prime. We define $\psi(x,y) = \#\{v \leq x, \text{ such that } v \text{ is } y\text{-smooth}\}$. It is known [22] that, for large x, the ratio $\psi(x, \sqrt[t]{x})/x$ is equivalent to Dickman's function defined by:

$$\rho(t) = \begin{cases} 1 & \text{if } \ 0 \leq t \leq 1 \\ \rho(n) - \displaystyle\int_n^t \frac{\rho(v-1)}{v} dv & \text{if } \ n \leq t \leq n+1 \end{cases}$$

$\rho(t)$ is thus an approximation of the probability that a u-bit number is $2^{u/t}$-smooth; Table 1 gives the numerical value of $\rho(t)$ (on a logarithmic scale) for $1 \leq t \leq 10$. The following theorem [12] gives an asymptotic estimate of the probability that an integer is smooth:

Theorem 1. *Let x be an integer and let $L_x[\beta] = \exp\left(\beta \cdot \sqrt{\log x \log\log x}\right)$. Let t be an integer randomly distributed between zero and x^γ for some $\gamma > 0$. Then for large x, the probability that all the prime factors of t are less than $L_x[\beta]$ is given by $L_x\left[-\gamma/(2\beta) + o(1)\right]$.*

Using this theorem, an asymptotic analysis of Desmedt and Odlyzko's attack is given in [17]. The analysis yields a time complexity of:

$$L_x[\sqrt{2} + o(1)]$$

where x is a bound on $\mu(m)$. This complexity is sub-exponential in the size of the integers $\mu(m)$. In practice, the attack is feasible only if the $\mu(m_i)$ are relatively small (*e.g.* less than 200 bits).

t	1	2	3	4	5	6	7	8	9	10
$-\log_2 \rho(t)$	0.0	1.7	4.4	7.7	11.5	15.6	20.1	24.9	29.9	35.1

Table 1. The value of Dickman's function for $1 \leq t \leq 10$.

4 Coron-Naccache-Stern's Attack

The Desmedt-Odlyzko's attack recalled in the previous section does not apply directly against ISO 9796-2, because in ISO 9796-2 the encoding function $\mu(m)$ is as long as the modulus N. Coron-Naccache-Stern's workaround [19] consisted in generating messages m_i such that a linear combination t_i of $\mu(m_i)$ and N is much smaller than N. Then, the attack can be applied to the integers t_i instead of $\mu(m_i)$.

More precisely, Coron *et al.* observed that it is sufficient to find a constant a and messages m_i such that:

$$t_i = a \cdot \mu(m_i) \bmod N$$

is small, instead of requiring that $\mu(m_i)$ is small. Namely, the factor a can be easily dealt with by regarding $a^{-1} \bmod N$ as an "additional factor" in $\mu(m_i)$; to that end we only need to add one more column in the matrix considered in Section 3. In their attack the authors used $a = 2^8$.

Obtaining a small $a \cdot \mu(m) \bmod N$ is done in [19] as follows. From the definition of ISO 9796-2:

$$\begin{aligned} \mu(m) &= \mathtt{6A}_{16} \quad \| \; m[1] \quad \| \; H(m) \quad \| \; \mathtt{BC}_{16} \\ &= \mathtt{6A}_{16} \cdot 2^{k-8} + m[1] \cdot 2^{k_h+8} + H(m) \cdot 2^8 + \mathtt{BC}_{16} \end{aligned}$$

where k is the modulus N size in bits, and k_h is the hash size. Euclidean division by N provides b and $0 \leq r < N < 2^k$ such that:

$$(\mathtt{6A}_{16} + 1) \cdot 2^k = b \cdot N + r$$

Denoting $N' = b \cdot N$ one can write:

$$\begin{aligned} N' &= \mathtt{6A}_{16} \cdot 2^k + \quad (2^k - r) \\ &= \mathtt{6A}_{16} \quad \| \; N'[1] \, \| \, N'[0] \end{aligned}$$

where N' is $k+7$ bits long and $N'[1]$ is $k - k_h - 16$ bits long, the same bit-length as $m[1]$. Consider now the linear combination:

$$\begin{aligned} t &= b \cdot N - a \; \cdot \mu(m) \\ &= N' \quad - 2^8 \cdot \mu(m) \end{aligned}$$

By setting $m[1] = N'[1]$ we get:

$$\begin{aligned} t = & \quad \mathtt{6A}_{16} \, \| \, N'[1] \, \| \, N'[0] \\ & - \mathtt{6A}_{16} \, \| \, m[1] \; \| \, H(m) \| \mathtt{BC00}_{16} \\ = & \quad \cancel{\mathtt{6A}_{16}} \, \| \, \cancel{N'[1]} \, \| \, N'[0] \\ & - \cancel{\mathtt{6A}_{16}} \, \| \, \cancel{N'[1]} \, \| \, H(m) \| \mathtt{BC00}_{16} \\ = & \qquad\qquad\qquad N'[0] - (H(m) \| \mathtt{BC00}_{16}) \end{aligned}$$

which gives $|t| \leq 2^{k_h+16}$. For $k_h = 160$, the integer t is therefore at most 176-bits long.

The forger can thus modify $m[2]$, and therefore $H(m)$, until he gets a set of messages whose t-values are B-smooth and express one such $\mu(m_\tau)$ as a multiplicative combination of the others. As per the analysis in [19], attacking the instances $k_h = 128$ and $k_h = 160$ requires (respectively) 2^{54} and 2^{61} operations.

5 Our new Attack

We improve the above complexities by using four new ideas: we accelerate Desmedt-Odlyzko's process using Bernstein's smoothness detection algorithm [7], instead of trial division; we also use the large prime variant [1]; moreover, we modify Coron *et al.*'s attack by selecting better messages and by optimizing exhaustive search to balance complexities.

5.1 Bernstein's Smoothness Detection Algorithm

The B-smooth part of an integer t is the product (with multiplicities) of all of its prime factors less or equal to B. In particular, an integer t is B-smooth if and only if its B-smooth part is equal to t.

Bernstein [7] describes the following algorithm for finding the B-smooth parts of each integer in a large list $\{t_1, \ldots, t_n\}$, and hence deduce, in particular, which of those integers are B-smooth.

Algorithm: Given the list of all prime numbers $p_1, \ldots, p_\ell$ up to B in increasing order, and a collection of positive integers $t_1, \ldots, t_n$, output the B-smooth part of each t_i:

1. Compute the product $z \leftarrow p_1 \times \cdots \times p_\ell$. This can be done in time and space $\widetilde{\mathcal{O}}(\ell)$ using a product tree.
2. Compute the modular reductions $z_1 \leftarrow z \bmod t_1, \ldots, z_n \leftarrow z \bmod t_n$ of z modulo each of the t_i's. This can again be done in quasilinear time in the size of the input using a remainder tree.
3. For each $i \in \{1, \ldots, n\}$: Compute $y_i \leftarrow (z_i)^{2^e} \bmod t_i$ by repeated squaring, where e is the smallest non-negative integer such that $2^{2^e} \geq t_i$.
4. For each $i \in \{1, \ldots, n\}$: output $s_i \leftarrow \gcd(t_i, y_i)$ as the B-smooth part of t_i.

The algorithm is correct since for each $i \in \{1, \ldots, n\}$:

$$y_i \equiv \prod_{j=1}^{\ell} p_j^{2^e} \pmod{t_i}$$

and hence, if we denote by $v_j(t_i)$ the p_j-adic valuation of t_i, we have:

$$s_i = \gcd(t_i, y_i) = \prod_{j=1}^{\ell} p_j^{\min(v_j(t_i), 2^e)} = \prod_{j=1}^{\ell} p_j^{v_j(t_i)}$$

in view of the choice of e, and this is clearly the B-smooth part of t_i.

In order to achieve a satisfactory time complexity, it is important to use efficient integer arithmetic and tree-based algorithms in Steps 1 and 2.

Indeed, a naive algorithm for the computation of the product $z = p_1 \times \cdots \times p_\ell$ would amount to $\ell - 1$ multiplications of integers of size close to to the size of z (namely $\widetilde{\mathcal{O}}(\ell)$ bits), and would thus require quadratic time even with quasilinear arithmetic. Instead, the tree-based approach consists in carrying out the $\ell/2$ products $p_1p_2, p_3p_4, \ldots$ between contiguous pairs of p_i's, which are numbers of size $\leq 2 \log \ell$; and then the $\ell/4$ products $(p_1p_2)(p_3p_4), (p_5p_6)(p_7p_8), \ldots$ between pairs of pairs, which are of size $\leq 4 \log \ell$; and so on until the whole product is obtained. The product tree has depth $\log_2 \ell$ and level k consists of $\ell/2^{k+1}$ multiplications of numbers of $2^{k+1} \log \ell$ bits, so that the overall complexity is quasilinear in ℓ.

Similarly, to compute the modular reductions of z modulo each of the t_i's, one does not carry out each of the n Euclidean divisions sequentially, which would take time $\widetilde{\mathcal{O}}(n\ell)$, but instead computes a product tree of the t_i's, and then carries out the Euclidean division of z by the product of the first half all t_i's on the one hand (that product is a node in the product tree), and by the product of the second half of all t_i's on the other hand (also a node in the product tree). This first level takes time $2 \times \widetilde{\mathcal{O}}(\ell + n\alpha/2)$, where α is the bitsize of the t_i's. Then, the remainder of the first division is divided by the product of the first quarter of all t_i's, and by the product of the second quarter, whereas the remainder of the second division is divided by the product of the third quarter of all t_i's, and by the fourth quarter, for a total time of $4 \times \widetilde{\mathcal{O}}(n\alpha/4)$. And so on and so forth until the leaves of the tree are reached, at which points one obtains all the remainders of z modulo the t_i's. Level k consists of 2^{k+1} Euclidean divisions by integers of $n\alpha/2^k$ bits, and there are $\log_2(n\alpha)$ levels, so that the overall complexity is quasilinear in $n\alpha$ (and separately ℓ, accounting for the first level).

As a result, Bernstein obtains the following theorem.

Theorem 2 (Bernstein). *The algorithm computes the B-smooth part of each integer t_i in time $\mathcal{O}(\beta \log^2 \beta \log\log \beta)$, where β is the number of input bits.*

In other words, given a list of n integers $t_i < 2^\alpha$ and the list of the first ℓ primes, the algorithm will detect the B-smooth t_i's, where $B = p_\ell$, in complexity:

$$\mathcal{O}(\beta \cdot \log^2 \beta \cdot \log\log \beta)$$

where $\beta = n \cdot \alpha + \ell \cdot \log_2 \ell$ is the total number of input bits. For large n and fixed α, ℓ, the asymptotic complexity is $\mathcal{O}(n \cdot \alpha \cdot \log^2 n \cdot \log\log n)$.

Optimization for large n. When n is very large, it becomes actually more efficient to run the algorithm k times, on batches of $n' = n/k$ integers. In the following we determine the optimal n' and the corresponding running time. We assume that for a single batch the algorithm runs in time:

$$\mathrm{BatchTime}(n', \alpha, \ell) = C \cdot \beta' \cdot \log^2 \beta' \cdot \log\log \beta'$$

where C is a constant and $\beta' = n' \cdot \alpha + u$ is the bit-length of the batch, where $u = \ell \cdot \log_2 \ell$ is the p_i-list's size in bits. The total running time is then:

$$\mathrm{TotalTime}(n, \alpha, \ell, n') = C \cdot \frac{n}{n'} \cdot \beta' \cdot \log^2 \beta' \cdot \log\log \beta'$$

The running time of a single batch only depends on β'. Hence, as a first approximation one could select an n' equating the sizes of the t_i-list and the p_i-list. This gives $n' \cdot \alpha = u$, and therefore $\beta' = 2n' \cdot \alpha$, which gives a total running time of $C \cdot 2n \cdot \alpha \cdot \log^2 \beta' \cdot \log\log \beta'$.

A more accurate analysis reveals that TotalTime is minimized for a slightly larger value of n'. Let $u = \ell \cdot \log_2 \ell$ and n' such that $n' \cdot \alpha = \gamma \cdot u$ for some parameter γ, which gives $\beta' = (\gamma + 1) \cdot u$, and:

$$\mathrm{TotalTime}(n, \alpha, \ell, \gamma) = C \cdot \frac{n \cdot \alpha}{u} \cdot \frac{\beta' \cdot \log^2 \beta' \cdot \log\log \beta'}{\gamma}$$

We look for the optimal γ. We neglect the $\log\log\beta'$ term and consider the function:

$$f(u,\gamma) = \frac{\beta' \cdot \log^2 \beta'}{\gamma} \quad \text{where} \quad \beta' = u \cdot (\gamma + 1)$$

Setting $\partial f(u,\gamma)/\partial\gamma = 0$, we get $u \cdot (\log^2 \beta' + 2\log\beta') \cdot \gamma - \beta' \log^2 \beta' = 0$, which gives $(\log b + 2) \cdot \gamma = (\gamma + 1)\log b$ and then $2\gamma = \log b$. This gives $2\gamma = \log u + \log(\gamma + 1)$, and neglecting the $\log(\gamma + 1)$ term, we finally get:

$$\gamma = (\log u)/2$$

as the optimal γ. This translates into running time as:

$$\text{TotalTime}(n, \alpha, \ell) \simeq C \cdot n \cdot \alpha \cdot \log^2 \beta' \cdot \log\log\beta' \tag{3}$$

where $\beta' \simeq (u \log u)/2$ and $u = \ell \cdot \log_2 \ell$.

Other optimizations. Bernstein recommends a number of speed-up ideas of which we used a few. In our experiments we used the *scaled remainder tree* [9], which replaces most division steps in the remainder tree by multiplications. This algorithm is fastest when FFT multiplications are done modulo numbers of the form $2^\alpha - 1$: we used this *Mersenne* FFT *multiplication* as well, as implemented in Gaudry, Kruppa and Zimmermann's GMP patch [24]. Other optimizations included computing the product z only once, and treating the prime 2 separately.

Bernstein's algorithm was actually the main source of the attack's improvement. It proved about 1000 faster than the trial division used in [19].

5.2 The Large Prime Variant

An integer is said to be *semi-smooth* with respect to y and z if its greatest prime factor is $\leq y$ and all other factors are $\leq z$. Bach and Peralta [1] define the function $\sigma(u,v)$, which plays for semi-smoothness the role played by Dickman's ρ function for smoothness: $\sigma(u,v)$ is the asymptotic probability that an integer n is semi-smooth with respect to $n^{1/v}$ and $n^{1/u}$.

In our attack we consider integers t_i which are semi-smooth with respect to B_2 and B, for some second bound B_2 such that $B < B_2 < B^2$. This is easy to detect using Bernstein's algorithm: for t_i to be (B_2, B)-semi-smooth, it suffices that its B-smooth part s_i (as computed by the algorithm above) satisfy $t_i/s_i \leq B_2$. Indeed, by definition, t_i/s_i has no prime factor smaller than B; therefore, if it is less or equal to $B_2 < B^2$ it must be prime itself (or equal to 1), and thus $t_i = s_i \cdot (t_i/s_i)$ is (B_2, B)-semi-smooth.

Namely it is often convenient in sieving algorithms for integer factorization and other problems (NFS, index calculus, etc.) to consider not only smooth numbers, which can be decomposed over the factor base, but also semi-smooth numbers, which cannot be decomposed directly, but do yield decomposable numbers when two or more are found corresponding to the same large prime: in other words, if t_1, t_2 are both (B_2, B)-semi-smooth, and the large primes t_1/s_1 and

t_2/s_2 are equal, then the rational number t_1/t_2 is B-smooth and can thus be considered in the relation-finding stage.

A detailed complexity analysis of this "large prime" variant in our context is provided in Appendix A.

5.3 Constructing Smaller $a \cdot \mu(m) - b \cdot N$ Candidates

In this paragraph we show how to construct smaller $t_i = a \cdot \mu(m_i) - b \cdot N$ values for ISO 9796-2. Smaller t_i-values increase smoothness probability and hence accelerate the forgery process.

We write:

$$\mu(x, h) = \mathtt{6A}_{16} \cdot 2^{k-8} + x \cdot 2^{k_h+8} + h \cdot 2^8 + \mathtt{BC}_{16}$$

where $x = m[1]$ and $h = H(m)$, with $0 < x < 2^{k-k_h-16}$.

We first determine $a, b > 0$ such that the following two conditions hold:

$$0 < b \cdot N - a \cdot \mu(0, 0) < a \cdot 2^{k-8} \tag{4}$$

$$b \cdot N - a \cdot \mu(0, 0) = 0 \pmod{2^8} \tag{5}$$

and a is of minimal size. Then by Euclidean division we compute x and r such that:

$$b \cdot N - a \cdot \mu(0, 0) = (a \cdot 2^{k_h+8}) \cdot x + r$$

where $0 \le r < a \cdot 2^{k_h+8}$ and using (4) we have $0 \le x < 2^{k-k_h-16}$ as required. This gives:

$$b \cdot N - a \cdot \mu(x, 0) = b \cdot N - a \cdot \mu(0, 0) - a \cdot x \cdot 2^{k_h+8} = r$$

Moreover as per (5) we must have $r = 0 \bmod 2^8$; denoting $r' = r/2^8$ we obtain:

$$b \cdot N - a \cdot \mu(x, h) = r - a \cdot h \cdot 2^8 = 2^8 \cdot (r' - a \cdot h)$$

where $0 \le r' < a \cdot 2^{k_h}$. We then look for smooth values of $r' - a \cdot h$, whose size is at most k_h plus the size of a.

If a and b are both 8-bit integers, this gives 16 bits of freedom to satisfy both conditions (4) and (5); heuristically each of the two conditions is satisfied with probability $\simeq 2^{-8}$; therefore, we can expect to find such an $\{a, b\}$ pair; we can enable slightly larger a and b if necessary. For example, for the RSA-2048 challenge, we found $\{a, b\}$ to be $\{625, 332\}$; therefore, for RSA-2048 and $k_h = 160$, the integer to be smooth is 170-bits long (instead of 176-bits in Coron *et al.*'s original attack). This decreases the attack complexity further. We provide in Table 2 the optimal $\{a, b\}$ pairs for for several RSA challenge moduli.

6 Attacking ISO 9796-2

We combined all the building-blocks listed in the previous section to compute an actual forgery for ISO 9796-2, with the RSA-2048 challenge modulus. The

Challenge	RSA-704	RSA-768	RSA-896	RSA-1024	RSA-1536	RSA-2048
a	481	251	775	311	581	625
b	228	132	412	172	316	332

Table 2. $\{a, b\}$ values for several RSA challenge moduli.

implementation replaced Coron *et al.*'s trial division by Bernstein's algorithm, replaced Coron *et al.*'s $a \cdot \mu(m) - b \cdot N$ values by the shorter t_i's introduced in paragraph 5.3 and took advantage of the large prime variant. Additional speed-up was obtained by exhaustive search for particular digest values.

As is usual for algorithms based on sieving methods, our attack can be roughly divided in two main stages: relation generation on the one hand, in which we try to generate many smooth and semismooth values t_i, yielding a large, sparse matrix of relations over our factor base, and linear algebra on the other hand, where we look for a non zero vector in the kernel of that large matrix, deducing a forgery. We provide technical details on both stages below.

6.1 Relation generation

Relation generation in our attacks amounted to computing many integers of the form $t_i = bN - a\mu(x, h_i)$ discussed in §5.3 (at most 170 bits long), and using Bernstein's algorithm to find the smooth and semismooth ones among them (with respect to suitable smoothness bounds). As shown in §5.1, Bernstein's algorithm is best applied on relatively small batches of such integers, and the whole relation generation process is thus an embarrassingly parallel problem.

As a result, we found it convenient to run this part of the attack on Amazon's EC2 cloud computing service, which also helps putting a simple dollar figure on the complexity of cryptanalytic attacks.

The Amazon grid. Amazon Web Services, Inc. offers virtualized computer instances for rent on a pay by the hour basis, which we found convenient to run our computations. At the time of our attack, the best-suited for CPU-intensive tasks featured 8 Intel Xeon 64-bit cores clocked at 2.4GHz supporting the Core2 instruction set, as well as 7GB RAM and 1.5TB disk space. Renting such a capacity cost US$0.80 per hour. One could launch up to 20 such instances in parallel, and possibly more subject to approval by Amazon (20 were enough for our purpose so we didn't apply for more).

When an instance is launched, it starts up from a disk image containing a customizable UNIX operating system. In the experiment, we ran a first instance using the basic Linux distribution provided by default, installed necessary tools and libraries, compiled our own programs and made a disk image containing our code, to launch subsequent instances with. When an instance terminates, its disk space is freed, making it necessary to save results to some permanent storage means. We simply transfered results to a machine of ours over the network.

Amazon also charges for network bandwidth but data transmission costs were negligible in our case.

All in all, we used about 1,100 instance running hours (including setup and tweaks) over a little more than two days. While we found the service to be rather reliable, one instance failed halfway through the computation, and its intermediate results were lost.

Parameter selection. The optimal choice of ℓ for 170 bits is about 2^{21}. Since the Amazon instances are memory-constrained (less than 1GB of RAM per core), we preferred to use $\ell = 2^{20}$. This choice had the additional advantage of making the final linear algebra step faster, which is convenient since this step was run on a single off-line PC. Computing the product of the first ℓ primes itself, as used in Bernstein's algorithm, was done once and for all in a matter of seconds using MPIR [26].

Hashing. Since smoothness detection part of the attack works on batches of t_i's (in our cases, we chose batches of 2^{19} integers), we had to compute digests of messages m_i in batches as well. The messages themselves are 2048-bit long, *i.e.* as long as N, and with the following structure: a constant 246-byte prefix followed by a 10-byte seed. The first two bytes identify a family of messages examined on a single core of one Amazon instance, and the remaining eight bytes are explored by increments of 1 starting from 0.

Messages were hashed using OpenSSL's implementation of SHA-1. For each message, we only need to compute one SHA-1 block, since the first three 64-byte blocks are fixed. This computation is relatively fast compared to Bernstein's algorithm, so we have a bit of leeway for exhaustive search. We can compute a large number of digests, keeping the ones likely to give rise to a smooth t_i. We did this by selecting digests for which the resulting t_i would have many zeroes as leading and trailing bits.

More precisely, we looked for a particular bit pattern at the beginning and at the end of each digest h_i, such that finding n matching bits results in n null bits at the beginning and at the end of t_i. The probability of finding n matching bits when we add the number of matches at the beginning and at the end is $(1+n/2)\cdot 2^{-n}$, so we expect to compute $2^n/(1+n/2)$ digests per selected message. We found $n = 8$ to be optimal: on average, we need *circa* 50 digests to find a match, and the resulting t_i is at most $170 - 8 = 162$ bit long (once powers of 2 are factored out).

Finding smooth and semi-smooth integers Once a batch of 2^{19} appropriate t_i's is generated, we factor out powers of 2, and feed the resulting odd numbers into our C++ implementation of Bernstein's algorithm. This implementation uses the MPIR multi-precision arithmetic library [26], which we chose over vanilla GMP because of a number of speed improvements, including J.W. Martin's patch for the Core2 architecture. We further applied Gaudry, Kruppa and Zimmermann's FFT patch, mainly for their implementation of *Mersenne* FFT multiplication, which is useful in the scaled remainder tree [9].

We looked for B-smooth as well as for (B, B_2)-semi-smooth t_i's, where $B =$ 16,290,047 is the 2^{20}-th prime, and $B_2 = 2^{27}$. Each batch took $\simeq 40$ seconds to generate and to process, and consumed about 500MB of memory. We ran 8 such processes in parallel on each instance to take advantage of the 8 cores, and 19 instances simultaneously.

After processing 647,901 such batches in roughly 1,100 CPU hours and a little over two days on the wall clock, we finally obtained sufficiently many relations for our purposes—namely 684,365 smooth t_i's and 366,302 collisions between 2,786,327 semi-smooth t_i's, for a total of 1,050,667 columns (slightly in excess of the $\ell = 2^{20} =$ 1,048,576 required).

6.2 Linear algebra

The output of the relation generation stage was a large, sparse matrix over GF(2), and all that remained to do to find a forgery was to find a non zero vector in its kernel. This was done in a few hours on a single desktop PC using an free software implementation of the block Wiedemann algorithm.

The exponent matrix. More precisely, as mentioned above, the exponent matrix was of size 1,048,576 × 1,050,667, and it had 14,215,602 non zero entries (13.5 per column on average, or 10^{-5} sparsity; the columns derived from the large prime variant tend to have twice as many non-zero entries, of course).

A number of rows contained only one non zero entry. As a preprocessing stage to the actual linear algebra computation, such rows and the corresponding columns could be safely removed, and that process was repeated recursively until no single entry remained. This resulted in a reduced matrix of dimension 750,031 × 839,908.

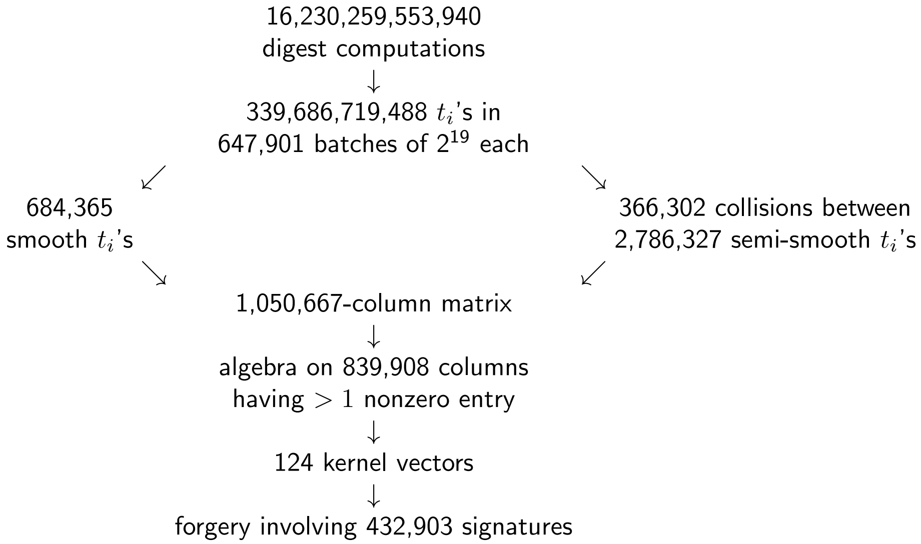
Block Wiedemann. We found non zero kernel elements of the final sparse matrix over GF(2) using Coppersmith's block Wiedemann algorithm [13] implemented in WLSS2 [34, 40], with parameters $m = n = 4$ and $\kappa = 2$. The whole computation took 16 hours on one 2.7GHz personal computer, with the first (and longest) part of the computation using 2 cores, and the final part using 4 cores.

The program obtained 124 kernel vectors with Hamming weights ranging from 337,458 to 339,641. Since columns obtained from pairs of semi-smooth numbers account for two signatures each, the number of signature queries required to produce the 124 corresponding forgeries is slightly larger, and ranges between 432,903 and 435,859.

Being written with the quadratic sieve in mind, the block Wiedemann algorithm in WLSS2 works over GF(2). There exist, however, other implementations for different finite fields.

6.3 Summary of the experiment

The entire experiment can be summarized as follows:



16,230,259,553,940 digest computations

339,686,719,488 t_i's in 647,901 batches of 2^{19} each

684,365 smooth t_i's

366,302 collisions between 2,786,327 semi-smooth t_i's

1,050,667-column matrix

algebra on 839,908 columns having > 1 nonzero entry

124 kernel vectors

forgery involving 432,903 signatures

7 Cost Estimates

The experiment described in the previous section can be used as a benchmark to estimate the cost of the attack as a function of the size of the t_i's, denoted α; this will be useful for analyzing the security of the EMV specifications, where α is bigger (204 bits instead of 170 bits).

$\alpha = \log_2 t_i$	$\log_2 \ell$	Estimated TotalTime	$\log_2 n$	Amazon EC2 cost (US$)
64	11	15 seconds	20	negligible
128	19	4 days	33	10
160	21	6 months	38	470
170	22	1.8 years	40	1,620
176	23	3.8 years	41	3,300
204	25	95 years	45	84,000
232	27	19 centuries	49	1,700,000
256	30	320 centuries	52	20,000,000

Table 3. Bernstein+Large prime variant. Estimated parameter trade-offs, running times and costs, for various t_i sizes, as of Spring 2009.

Results are summarized in Table 3. We assume that the t_i's are uniformly distributed α-bit integers and express costs as a function of α. We only take into account the running time of the smoothness detection algorithm from Section 5.1, and do not include the linear algebra step whose computational requirements are very low compared to the smoothness detection step. The running times are extrapolated from the experiments performed in the previous section, using Equation (3), where the total number of messages n to be examined is estimated as

$$n \simeq \frac{\ell}{\rho(\alpha/\log_2 p_\ell)}$$

where ρ is Dickman function and $p_\ell \simeq \ell \log \ell$; for simplicity we don't consider the large prime variant. For each value of α, we compute the optimal value of ℓ that minimizes the running time. The number of signatures required for the forgery is then $\tau = \ell + 1$. Note that in Table 3 we do not assume any exhaustive search on the t_i's; this is why the cost estimate for $\alpha = 170$ in Table 3 is about the double of the cost of our experimental ISO 9796-2 forgery.

Running times are given for a single 2.4GHz PC. Costs correspond to the Amazon EC2 grid as of Spring 2009, as in the previous section. Estimates show that the attack is feasible up to $\simeq$ 200 bits, but becomes infeasible for larger values of α. We also estimate $\log_2 n$, where n is the total number of messages to be examined.

7.1 Fewer Queries

The number of signatures actually used by the forger is not τ but the number of nonzero β_i values in the formula:

$$\mu(m_\tau) = \left(\prod_{j=1}^{\ell} p_j^{\gamma_j} \right)^e \cdot \prod_{i=1}^{\tau-1} \mu(m_i)^{\beta_i}$$

Assuming that $(\beta_1, \ldots, \beta_{\tau-1})$ is a random vector of $\mathbb{Z}_e^{\tau-1}$ only $\tau(e-1)/e$ of the signatures will be actually used to compute the forgery. The gain is significant when e is a very small exponent (*e.g.* 2 or 3). However, one can try to generate more than τ candidates but select the subset of signatures minimizing the number of nonzero β_i values. Such a sparse β-vector may allow to reduce the number of queries and defeat ratification counters meant to restrict the number of authorized signature queries.

In essence, we are looking at a random $[\ell, k]$ code: a kernel vector has ℓ components which, for $e = 2$, can be regarded as a set of independent unbiased Bernoulli variables. The probability that such a vector has weight less than $w = \sum_{i=1}^{\tau-1} \beta_i$ is thus:

$$\sum_{j=1}^{w} \binom{\ell}{j} 2^{-\ell} \simeq \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{w - \ell/2}{\sqrt{\ell/2}}\right)\right)$$

We have 2^k such vectors in the kernel, hence the probability that at least one of them has a Hamming weight smaller than w is surely bounded from above by:

$$2^k \times \frac{1}{2}\left(1+\operatorname{erf}\left(\frac{w-\ell/2}{\sqrt{\ell/2}}\right)\right) = 2^{k-1}\left(1+\operatorname{erf}\left(\frac{w-\ell/2}{\sqrt{\ell/2}}\right)\right)$$

Let c denote the density bias of w *i.e.*, $w = (1/2 - c)\ell$. The previous bound becomes:

$$\begin{aligned} p(c) &= 2^{k-1}\left(1+\operatorname{erf}\left(-c\sqrt{2\ell}\right)\right) = 2^{k-1}\left(1-\operatorname{erf}\left(c\sqrt{2\ell}\right)\right) \\ &= 2^{k-1}\operatorname{erfc}(c\sqrt{2\ell}) \underset{\ell\to+\infty}{\sim} \frac{2^{k-1}\exp(-2\ell c^2)}{c\sqrt{2\pi\ell}} \end{aligned}$$

For $\ell = 2^{20}$, even if we take k as large as 2^{10} (the largest subspace dimension considered tractable, even in much smaller ambient spaces), we get $p(1/50) \simeq 10^{-58}$, so the probability that there exists a kernel vector of weight $w < 500{,}000$ is negligible. In addition, even if such a vector existed, techniques for actually computing it, *e.g.* [10], seem to lag far behind the dimensions we deal with.

It follows that a better strategy to diminish w is to simply decrease ℓ. The expected payoff might not be that bad: If the attacker is limited to, say, 2^{16} signatures, then he can pick $\ell = 2^{17}$, and for 196-bit numbers (204 bits minus 8 bits given by exhaustive search), the attack becomes about 15 times slower than the optimal choice, $\ell = 2^{24}$ (note as well that more exhaustive search becomes possible in that case). That is slow, but perhaps not excruciatingly so.

8 Application to EMV Signatures

EMV is a collection of industry specifications for the inter-operation of payment cards, POS terminals and ATMs. The EMV specifications [23] rely on ISO 9796-2 signatures to certify public-keys and to authenticate data. For instance, when an Issuer provides application data to a Card, this data must be signed using the Issuer's private key S_{I}. The corresponding public-key P_{I} must be signed by a Certification Authority (CA) whose public-key is denoted P_{CA}. The signature algorithm is RSA with $e = 3$ or $e = 2^{16} + 1$. The bit length of all moduli is always a multiple of 8.

EMV uses special message formats; 7 different formats are used, depending on the message type. In the following we describe one of these formats: the *Static Data Authentication, Issuer Public-key Data* (SDA-IPKD), and adapt our attack to it.

8.1 EMV Static Data Authentication, Issuer Public-key Data (SDA-IPKD)

We refer the reader to §5.1, Table 2, page 41 in EMV [23]. SDA-IPKD is used by the CA to sign the issuer's public-key P_{I}. The message to be signed is as follows:

$$m = \mathsf{02}_{\mathsf{16}} \| X \| Y \| N_{\text{I}} \| \mathsf{03}_{\mathsf{16}}$$

where X represents 6 bytes that can be controlled by the adversary and Y represents 7 bytes that cannot be controlled. N_{I} is the Issuer's modulus to be certified. More precisely, $X = \text{ID}\|\text{DATE}$ where ID is the issuer identifier (4 bytes) and DATE is the *Certificate Expiration Date* (2 bytes); we assume that both can be controlled by the adversary. $Y = \text{CSN}\|C$ where CSN is the 3-bytes *Certificate Serial Number* assigned by the CA and C is a constant. Finally, the modulus to be certified N_{I} can also be controlled by the adversary.

With ISO 9796-2 encoding, this gives:

$$\mu(m) = \texttt{6A02}_{16}\|X\|Y\|N_{\text{I},1}\|H(m)\|\texttt{BC}_{16}$$

where $N_{\text{I}} = N_{\text{I},1}\|N_{\text{I},2}$ and the size of $N_{\text{I},1}$ is $k - k_h - 128$ bits. k denotes the modulus size and $k_h = 160$ as in ISO 9796-2.

8.2 Attacking SDA-IPKD

To attack SDA-IPKD write:

$$\mu(X, N_{\text{I},1}, h) = \texttt{6A02}_{16} \cdot 2^{k_1} + X \cdot 2^{k_2} + Y \cdot 2^{k_3} + N_{\text{I},1} \cdot 2^{k_4} + h$$

where Y is constant and $h = H(m)\|\texttt{BC}_{16}$. We have:

$$\begin{cases} k_1 = k \quad - 16 \\ k_2 = k_1 - 48 = k - 64 \\ k_3 = k_2 - 56 = k - 120 \\ k_4 = k_h + \ 8 \ = 168 \end{cases}$$

Generate a random k_a-bit integer a, where $36 \leq k_a \leq 72$, and consider the equation:

$$b \cdot N - a \cdot \mu(X, 0, 0) = b \cdot N - a \cdot X \cdot 2^{k_2} - a \cdot (\texttt{6A02}_{16} \cdot 2^{k_1} + Y \cdot 2^{k_3})$$

If we can find integers X and b such that $0 \leq X < 2^{48}$ and:

$$0 \leq b \cdot N - a \cdot \mu(X, 0, 0) < a \cdot 2^{k_3} \tag{6}$$

then as previously we can compute $N_{\text{I},1}$ by Euclidean division:

$$b \cdot N - a \cdot \mu(X, 0, 0) = (a \cdot 2^{k_4}) \cdot N_{\text{I},1} + r \tag{7}$$

where $0 \leq N_{\text{I},1} < 2^{k_3 - k_4}$ as required and $0 \leq r < a \cdot 2^{k_4}$, which gives:

$$b \cdot N - a \cdot \mu(X, N_{\text{I},1}, h) = r - a \cdot h$$

and therefore $|b \cdot N - a \cdot \mu(X, N_{\text{I},1}, h)| < a \cdot 2^{k_4}$ for all values of h.

In the above we assumed Y to be a constant. Actually the first 3 bytes of Y encode the CSN assigned by the CA, and may be different for each new certificate (see Appendix B). However if the attacker can predict the CSN, then he can compute a different a for every Y and adapt the attack by factoring a into a product of small primes.

Finding small X and b so as to minimize the value of

$$|b \cdot N - a \cdot X \cdot 2^{k_2} - a \cdot (\mathtt{6A02}_{\mathtt{16}} \cdot 2^{k_1} + Y \cdot 2^{k_3})|$$

is a Closest Vector Problem (CVP) in a bi-dimensional lattice; a problem that can be easily solved using the LLL algorithm [36]. We first determine heuristically the minimal size that can be expected; we describe the LLL attack in Appendix B.

Since $a \cdot \mathtt{6A02}_{\mathtt{16}} \cdot 2^{k_1}$ is an $(k + k_a)$-bit integer, with $X \simeq 2^{48}$ and $b \simeq 2^{k_a}$, heuristically we expect to find X and b such that:

$$0 \leq b \cdot N - a \cdot \mu(X, 0, 0) < 2^{(k+k_a)-48-k_a} = 2^{k-48} \simeq a \cdot 2^{k-48-k_a} = a \cdot 2^{k_3+72-k_a}$$

which is $(72 - k_a)$-bit too long compared to condition (6). Therefore, by exhaustive search we will need to examine roughly 2^{72-k_a} different integers a to find a pair (b, X) that satisfies (6); since a is k_a-bits long, this can be done only if $72 - k_a \leq k_a$, which gives $k_a \geq 36$. For $k_a = 36$ we have to exhaust the 2^{36} possible values of a.

Once this is done we obtain from (7):

$$t = b \cdot N - a \cdot \mu(X, N_{\textsc{i},1}, h) = r - a \cdot h$$

with $0 \leq r < a \cdot 2^{k_4}$. This implies that the final size of t values is $168 + k_a$ bits. For $k_a = 36$ this gives 204 bits (instead of 170 bits for plain ISO 9796-2). The attack's complexity will hence be higher than for plain ISO 9796-2.

In Appendix B we exhibit concrete (a, b, X) values for $k_a = 52$ and for the RSA-2048 challenge; this required $\simeq 2^{23}$ trials (109 minutes on a single PC). We estimate that for $k_a = 36$ this computation will take roughly 13 years on a single PC, or equivalently US\$11,000 using the EC2 grid.

9 Conclusion

We have described an improved attack against the amended version of ISO 9796-2, that is for $k_h = 160$. The new attack applies to EMV signatures as well. Our new attack is similar to Coron *et al.* forgery but using Bernstein's smoothness detection algorithm instead of trial division. In practice we were able to compute a forgery for ISO 9796-2 in only two days, using a few dozens of servers on the Amazon EC2 grid, for a total cost of US\$800.

In response to this attack, the ISO 9796-2 standard was amended [31] to discourage the use of the ad-hoc signature padding in contexts where chosen-message attacks are an issue.

References

1. E. Bach and R. Peralta, *Asymptotic semismoothness probabilities*, Mathematics of Computation, vol. 65, number 216, 1996, pp. 1701–1715.

2. M. Bellare and P. Rogaway, *Random oracles are practical: A paradigm for designing efficient protocols*, Proceedings of CCS 1993, ACM, 1993, pp. 62–73.
3. M. Bellare and P. Rogaway, *Optimal Asymmetric Encryption: How to encrypt with RSA*, Proceedings of Eurocrypt 1994, LNCS, vol. 950, Springer-Verlag, 1995, pp. 92–111.
4. M. Bellare and P. Rogaway, *The Exact security of digital signatures: How to sign with RSA and Rabin*, Proceedings of Eurocrypt 1996, LNCS, vol. 1070, Springer-Verlag, 1996, pp. 399–416.
5. D.J. Bernstein and T. Lange (editors), *eBACS: ECRYPT Benchmarking of cryptographic systems*, `bench.cr.yp.to`.
6. D.J. Bernstein, *Fast Multiplications and its applications*, Algorithmic Number Theory, vol. 44, 2008.
7. D.J. Bernstein, *How to find smooth parts of integers*, 2004/05/10, `cr.yp.to/papers.html\#smoothparts`.
8. D.J. Bernstein, *Proving tight security for Rabin-Williams signatures*. Proceedings of Eurocrypt 2008, LNCS, vol. 4665, Springer-Verlag, 2008, pp. 70–87.
9. D.J. Bernstein, *Scaled remainder trees*, 2004/08/20, `cr.yp.to/papers.html\#scaledmod`.
10. D.J. Bernstein, T. Lange and Ch. Peters, *Attacking and defending the McEliece cryptosystem*, Proceedings of Post-Quantum Cryptography 2008, LNCS, vol. 5299, Springer-Verlag, 2008, pp. 31–46.
11. D. Bleichenbacher, *Chosen ciphertext attacks against protocols based on the RSA encryption standard*, Proceedings of Crypto 1998, LNCS, vol. 1462, Springer-Verlag, 1998, pp. 1–12.
12. E.R. Canfield, P. Erdős and C. Pomerance, *On a Problem of Oppenheim concerning 'Factorisation Numerorum'*, Journal of Number Theory, vol. 17, 1983, pp. 1–28.
13. D. Coppersmith, *Solving homogeneous linear equations over* GF(2) *via block Wiedemann algorithm*, Mathematics of Computation, vol. 62, number 205, 1994, pp. 333–350.
14. D. Coppersmith, J.-S. Coron, F. Grieu, S. Halevi, C.S. Jutla, D. Naccache and J.P. Stern, *Cryptanalysis of ISO 9796-1*, Journal of Cryptology, vol. 21, 2008, pp. 27–51.
15. D. Coppersmith, S. Halevi and C. Jutla, ISO 9796-1 *and the new, forgery strategy*, Research contribution to P.1363, 1999, `grouper.ieee.org/groups/1363/Research`.
16. J.-S. Coron, *Security proofs for partial domain hash signature schemes*, Proceedings of Crypto 2002, LNCS, vol. 2442, Springer-Verlag, 2002, pp. 613–626.
17. J.-S. Coron, Y. Desmedt, D. Naccache, A. Odlyzko and J.P. Stern, *Index calculation attacks on RSA signature and encryption* Designs, Codes and Cryptography, vol. 38, number 1, 2006, pp. 41–53.
18. J.-S. Coron, D. Naccache, M. Joye and P. Paillier, *New attacks on* PKCS*#1 v1.5 encryption*, Proceedings of Eurocrypt 2000, LNCS, vol. 1807, Springer-Verlag, 2000, pp. 369–381.
19. J.-S. Coron, D. Naccache and J.P. Stern, *On the security of RSA padding*, Proceedings of Crypto 1999, LNCS, vol. 1666, Springer-Verlag, 1999, pp. 1–18.
20. R.E. Crandall, E.W. Mayer and J.S. Papadopoulos, *The twenty-fourth Fermat number is composite*, Mathematics of Computation, volume 72, number 243, July 2003, pp. 1555–1572.
21. Y. Desmedt and A. Odlyzko. *A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes*, Proceedings of Crypto 1985, LNCS, vol. 218, Springer-Verlag, 1986, pp. 516–522.

22. K. Dickman, *On the frequency of numbers containing prime factors of a certain relative magnitude*, Arkiv för matematik, astronomi och fysik, vol. 22A, no. 10, 1930, pp. 1–14.
23. EMV, *Integrated circuit card specifications for payment systems*, Book 2. Security and Key Management. Version 4.2. June 2008. `www.emvco.com`.
24. P. Gaudry, A. Kruppa and P. Zimmermann, *A gmp-based implementation of Schőnhage-Strassen's large integer multiplication algorithm*, in Proceedings of ISSAC 2007, Waterloo, Ontario, Canada, ACM Press, 2007, pp. 167–174.
25. F. Grieu, *A chosen messages attack on the ISO/IEC 9796-1 signature scheme*, Proceedings of Eurocrypt 2000, LNCS, vol. 1807, Springer-Verlag, 2000, pp. 70–80.
26. W.B. Hart *et al.*, *Multiple Precision Integers and Rationals*, `www.mpir.org`.
27. W.B. Hart, D. Harvey *et al.*, *Fast Library for Number Theory*, `www.flintlib.org`.
28. ISO/IEC 9796, *Information technology – Security techniques – Digital signature scheme giving message recovery, Part 1: Mechanisms using redundancy*, 1999.
29. ISO 9796-2, *Information technology – Security techniques – Digital signature scheme giving message recovery, Part 2: Mechanisms using a hash-function*, 1997.
30. ISO 9796-2:2002, *Information technology – Security techniques – Digital signature schemes giving message recovery – Part 2: Integer factorization based mechanisms*, 2002.
31. ISO 9796-2:2010, *Information technology – Security techniques – Digital signature schemes giving message recovery – Part 2: Integer factorization based mechanisms*, 2010.
32. A. Joux, D. Naccache, E. Thomé, *When e-th roots become easier than factoring*, Proceedings of Asiacrypt 2007, LNCS, vol. 4833, Springer-Verlag, 2007, pp. 13–28.
33. B. Kaliski, *PKCS#1: RSA Encryption Standard, Version 1.5*, RSA Laboratories, November 1993.
34. E. Kaltofen and A. Lobo, *Distributed matrix-free solution of large sparse linear systems over finite fields*, Algorithmica, vol. 24, 1999, pp. 331–348.
35. C. Lanczos, *An iterative method for the solution of the eigenvalue problem of linear differential and integral operator*, Journal of Research of the National Bureau of Standards, vol. 45, 1950, pp. 255–282.
36. A.K. Lenstra, H.W. Lenstra, Jr., and L. Lovász, *Factoring polynomials with rational coefficients*. Mathematische Annalen, vol. 261, 1982, pp. 513–534.
37. A.K. Lenstra and H.W. Lenstra, Jr., *The Development of the number field sieve*, Berlin: Springer-Verlag, 1993.
38. H. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics, vol. 126, number 2, 1987, pp. 649–673.
39. Y. Liu, T. Kasper, K. Lemke-Rust, C. Paar: *E-Passport: Cracking basic access control keys*, OTM Conferences (2), 2007, pp. 1531-1547.
40. A. Lobo, *WLSS2: an implementation of the homogeneous block Wiedemann algorithm*, `www4.ncsu.edu/\~kaltofen/software/wiliss`.
41. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of applied cryptography*, CRC press, 1996.
42. M. Mezzarobba, *de auditu*, March 2009.
43. J.-F. Misarsky, *How (not) to design RSA signature schemes*, Proceedings of Public Key Cryptography 1998, LNCS, vol. 1431, Springer-Verlag, 1998, pp. 14–28.
44. P.L. Montgomery, *A block Lanczos algorithm for finding dependencies over* GF(2), Proceedings of Eurocrypt 1995, LNCS, vol. 921, Springer-Verlag, 1995, pp. 106–120.
45. NVIDIA, CUDA *Zone – The resource for* CUDA *developers*, `www.nvidia.com/cuda`.
46. D.A. Osvik, *de auditu*, March 2009.

47. C. Paar and M. Schimmer, COPACOBANA: *A Codebreaker for* DES *and other ciphers*, `www.copacobana.org`.
48. The PARI Group, *PARI/GP*, version 2.3.4, Bordeaux, 2008, `pari.math.u-bordeaux.fr`.
49. C. Pomerance, *The quadratic sieve factoring algorithm*, Proceedings of Eurocrypt 1984, LNCS, vol. 209, Springer-Verlag, 1985, pp. 169–182.
50. R. Rivest, A. Shamir and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Communications of the ACM, vol. 21, 1978, pp. 120–126.
51. The SAGE development team, SAGE *mathematics software (Version 3.3)*, 2009, `www.sagemath.org`.
52. D. Stinson, *Cryptography: theory and practice*, 3-rd edition, CRC Press, 2005.
53. M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D.A. Osvik, B. de Weger: *Short chosen-prefix collisions for* MD5 *and the creation of a rogue* CA *certificate*, Cryptology ePrint Archive, Report 2009/111, 2009.
54. V. Shoup, *Number Theory C++ Library (*NTL*) version 5.3.1.* `www.shoup.net/ntl`.

A Large Prime Variant: Complexity Analysis

In this appendix we provide an accurate analysis of the large prime variant in the context of our attack.

Assume that we check our t_i-list for (B, B_2)-semi-smoothness (instead of B-smoothness) and detect η semi-smooth numbers. Amongst those, we expect to find $\eta\lambda$ numbers that are actually B-smooth, for some $\lambda \in [0, 1]$ that can be expressed in terms of ρ and σ functions. If we further assume that the $\eta(1 - \lambda)$ remaining numbers, which are semi-smooth but non-smooth, have their largest prime factors uniformly distributed amongst the h primes between B and B_2, we expect to find about $\eta^2(1 - \lambda)^2/(2h)$ "collisions" between them, that is, about $\eta^2(1 - \lambda)^2/(2h)$ pairs of numbers with the same largest prime factor.

Note that:

$$h \simeq \frac{B_2}{\log B_2} - B$$

Let ℓ be the number of primes less than B. The smooth numbers in the list yield a total of $\eta\lambda$ exponent vectors over the first ℓ primes, and each of the collisions between the remaining semi-smooth numbers yields such an additional exponent vector. Since we need (slightly more than) ℓ vectors to forge a signature, we should examine enough t_i's to find η semi-smooth numbers, where η satisfies:

$$\ell = \eta\lambda + \frac{\eta^2(1 - \lambda)^2}{2h}$$

Solving for η, we get:

$$\eta = \frac{2\ell}{\lambda + \sqrt{2\ell \cdot (1 - \lambda)^2/h + \lambda^2}}$$

The probability β that a random α-bit integer is semi-smooth with respect to B_2 and $B \simeq \ell \cdot \log \ell$ is:

$$\beta = \sigma\left(\frac{\alpha \log 2}{\log(\ell \log \ell)}, \frac{\alpha \log 2}{\log B_2}\right)$$

and if γ denotes the probability that a random α-bit integer is B-smooth, we have:

$$\lambda = \frac{\gamma}{\beta} = \rho\left(\frac{\alpha \log 2}{\log(\ell \log \ell)}\right) / \sigma\left(\frac{\alpha \log 2}{\log(\ell \log \ell)}, \frac{\alpha \log 2}{\log B_2}\right)$$

In this large prime variant, we only need to generate $n' = \eta/\beta$ numbers to find enough exponent vectors, as opposed to $n = \ell/\gamma$ previously. Therefore, the large prime variant improves upon simple smoothness by a factor of roughly:

$$\vartheta = \frac{n}{n'} = \frac{\ell/\gamma}{\eta/\beta} = \frac{1}{\lambda} \cdot \frac{\ell}{\eta} = \frac{1}{2}\left[1 + \sqrt{1 + \frac{2\ell}{h}\left(\frac{1}{\lambda} - 1\right)}^{2}\right] \geq 1 \qquad (8)$$

ϑ is always greater than 1, and for the sizes we are interested in, say $100 \leq \alpha \leq 200$, we find $\vartheta \simeq 1.5$ for the best choice of B, and $B_2 \gtrsim 7B$. The reader is referred to Table 4 for precise figures.

Integer size α	128	144	160	176	192
Optimal $\log_2(\ell)$	19	20	21	23	24
Best ϑ	1.43	1.46	1.49	1.43	1.45

Table 4. Improvement factor ϑ due to the large prime variant.

According to formula (8), ϑ increases until B_2 reaches $\simeq 7B$, and decreases slowly thereafter. This is actually *not* the case: finding a larger t_i population to be semi-smooth can only produce *more* collisions. The decrease suggested by formula (8) stems from the assumption that the largest prime factors of the t_i's are uniformly distributed amongst the h primes between B and B_2, which is only approximately true. The imprecision grows with h (a larger B_2 doesn't spread the largest prime factors more thinly). Choosing a very large B_2 is not advisable, however, because it produces considerable extra output (searching for collisions becomes cumbersome) with negligible returns in terms of actual collisions. In the practical attack, we selected $\ell = 2^{20}$ and $B_2 = 2^{27} \simeq 9B$.

B LLL Attack on EMV SDA-IPKD Encoding

B.1 The LLL Attack

Given a, N we must minimize the value of:

$$\left| b \cdot N - a \cdot X \cdot 2^{k_2} - a \cdot (\mathtt{6A02}_{16} \cdot 2^{k_1} + Y \cdot 2^{k_3}) \right|$$

We show how this can be done using LLL. We write:

$$\begin{aligned} u &= a \cdot 2^{k_2} \\ v &= a \cdot (\texttt{6A02}_{\texttt{16}} \cdot 2^{k_1} + Y \cdot 2^{k_3}) \end{aligned}$$

where $N \simeq 2^k$, $X \simeq 2^{48}$, $a \simeq 2^{k_a}$, $u \simeq 2^{k-64+k_a}$ and $v \simeq 2^{k+k_a}$.

Hence we must minimize the absolute value of:

$$t = b \cdot N - x \cdot u - v$$

Consider the lattice of column vectors:

$$L = \begin{bmatrix} & & 2^{k-48} \\ & 2^{k-96} & \\ N & -u & -v \end{bmatrix}$$

As seen previously, heuristically, we can obtain $t \simeq 2^{k-48}$; therefore the coefficients in L are chosen so as to obtain a short vector of norm $\simeq 2^{k-48}$. More precisely, we look for a short column vector $\boldsymbol{c} \in L$ of the form:

$$\boldsymbol{c} = \begin{bmatrix} 2^{k-48} \\ x \cdot 2^{k-96} \\ b \cdot N - u \cdot x - v \end{bmatrix}$$

Theorem 3 (LLL). *Let L be a lattice spanned by $(u_1, \ldots, u_\omega)$. The LLL algorithm, given the vectors $(u_1, \ldots, u_\omega)$, finds in polynomial time a vector b_1 such that:*

$$\|b_1\| \leq 2^{(\omega-1)/4} \det(L)^{1/\omega}$$

Therefore, using LLL we can find a short vector of norm:

$$\|b_1\| \leq 2 \cdot (\det L)^{1/3} \leq 2 \cdot (2^{3k-144})^{1/3} \leq 2^{k-47}$$

Heuristically we hope that $b_1 = \boldsymbol{c}$, which allows solving for the values of b and X. The attack is heuristic but it works very well in practice, as shown in the next section.

B.2 Practical Value for EMV SDA-IPKD

Consider again the SDA-IPKD EMV format; we write:

$$\mu(X, N_{\text{I},1}, h) = \texttt{6A02}_{\texttt{16}} \cdot 2^{k_1} + X \cdot 2^{k_2} + Y \cdot 2^{k_3} + N_{\text{I},1} \cdot 2^{k_4} + h$$

where the constant Y is taken to be:

$$Y = \texttt{010203 0101 F8 01}_{\texttt{16}}$$

The first 3 bytes correspond to the CSN assigned by the CA (we took $\texttt{010203}_{\texttt{16}}$), $\texttt{0101}_{\texttt{16}}$ corresponds to the *hash algorithm indicator* and to the *public-key algorithm indicator*. $\texttt{F8}_{\texttt{16}} = 248$ is the issuer public-key length (in bytes) and $\texttt{01}_{\texttt{16}}$ is the length of the public exponent ($e = 3$).

Taking the RSA-2048 challenge for N, we have run the attack of the previous section for $k_a = 52$ and found the following values after $8{,}303{,}995 \simeq 2^{23}$ iterations:

$$a = 4127135343129068 \qquad b = 2192055331476458 \qquad X = 66766242156276$$

which are such that $0 < X < 2^{48}$ and:

$$0 \leq b \cdot N - a \cdot \mu(X, 0, 0) < a \cdot 2^{k_3} \tag{9}$$

as required.

The computation took $\simeq 109$ minutes on a single 2GHz PC. Therefore, for $k_a = 36$ we expect that 2^{36} trials to yield a triple $\{a, b, X\}$ satisfying condition (9) such that $|a| \leq 2^{36}$, within a running time of $\simeq 109 \cdot 2^{36-20} = 4.3 \cdot 10^8$ minutes = 13 years on a single PC, or equivalently for US\$11,000 using the EC2 grid.