Batch Verification of ECDSA Signatures

Sabyasachi Karati
Abhijit Das
Dipanwita Roychowdhury
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India
skarati,abhij,drc@cse.iitkgp.ernet.in

Bhargav Bellur
Debojyoti Bhattacharya
Aravind Iyer
General Motors Technical Centre India
India Science Lab, Bangalore, India
bhargav_bellur@yahoo.com
Debojyoti.bhattacharya@gmail.com
aravind.iyer@gm.com

Abstract. In this paper, we study several algorithms for batch verification of ECDSA signatures. The first of these algorithms is based upon the naive idea of taking square roots in the underlying field. We also propose two new and efficient algorithms which replace square-root computations by symbolic manipulations. Experiments carried out on NIST prime curves demonstrate a maximum speedup of above six over individual verification if all the signatures in the batch belong to the same signer, and a maximum speedup of about two if the signatures in the batch belong to different signers, both achieved by a fast variant of our second symbolic-manipulation algorithm. In terms of security, all the studied algorithms are equivalent to standard ECDSA* batch verification. These algorithms are practical only for small ($\leqslant 8$) batch sizes. To the best of our knowledge, this is the first reported study on the batch verification of *original* ECDSA signatures.

Keywords: Digital Signatures, Elliptic Curves, ECDSA, ECDSA*, Batch Verification, Modular Square Root, Symbolic Computation, Linearization.

1 Introduction

Batch verification is used to verify multiple digital signatures in time less than total individual verification time. The concept of batch verification is introduced by Naccache et al [7] in EuroCrypt'94. They propose an interactive batch-verification protocol for DSA [8]. In this protocol, the signer generates t signatures through interaction with the verifier, and then the verifier validates all these t signatures simultaneously.

Harn, in 1998, proposes an efficient scheme [4, 5] for the batch verification of RSA signatures [12], where multiple signatures signed by the same private key can be verified simultaneously. Harn's scheme uses only one exponentiation for batches of any size t. There are some weaknesses in this scheme. For example, if batch verification

fails, we cannot identify the faulty signature(s) without making individual verification. Moreover, Harn's scheme does not adapt to the case of signatures from multiple signers.

These protocols are not straightaway applicable to ECDSA signatures [2, 6]. Since ECDSA requires smaller key and signature sizes than DSA and RSA, there has been a growing interest in ECDSA. ECDSA* [1], a modification of ECDSA, permits an easy adaptation of Naccache et al's batch-verification protocol for DSA. Cheon and Yi [3] study batch verification of ECDSA* signatures, and report speedup factors of up to 7 for same signer and 4 for different signers. However, ECDSA* is not a standard, and is thus unacceptable, particularly in applications where interoperability is of concern. More importantly, ECDSA* increases the signature size compared to ECDSA without any increase in the security. Consequently, batch verification of original ECDSA signatures turns out to be a practically important open research problem. To the best of our knowledge, no significant result in this area has ever been reported in the literature.

In this paper, we propose three algorithms to verify *original* ECDSA signatures in batches. Our algorithms apply to all cases of ECDSA signatures sharing the same curve parameters, although we obtain good speedup figures when all the signatures in the batch come from the same signer. Our algorithms are effective only for small batch sizes (like $t \leqslant 8$). The first algorithm we introduce (henceforth denoted as Algorithm N) is based upon a naive approach of taking square roots in the underlying field. As the field size increases, square-root computations become quite costly. We modify Algorithm N by replacing square-root calculations by symbolic manipulations. We propose two ECDSA batch-verification algorithms, called S1 and S2, using symbolic manipulations. Algorithm S1 is not very practical, but is discussed in this paper, for it provides the theoretical and practical foundations for arriving at Algorithm S2. For a wide range of field and batch sizes, Algorithm S2 convincingly outperforms the naive Algorithm N. Both S1 and S2 are probabilistic algorithms in the Monte Carlo sense, that is, they may occasionally fail to verify correct signatures. We analytically establish that for randomly generated signatures, the failure probability is extremely low.

The rest of this paper is organized as follows. In Section 2, we identify the problems associated with ECDSA batch verification. In this process, we introduce the ECDSA signature scheme, and set up the notations which we use throughout the rest of the paper. In Section 3, we introduce a naive batch-verification algorithm N and its variant N$'$. Section 4 elaborates our new algorithm S1 based upon symbolic manipulations. Section 5 presents an analytic study of Algorithm S1. We furnish details about the running time, the cases of failure, and the security of Algorithm S1. The running time estimates for Algorithm S1 indicate that this algorithm is expected to perform poorly unless the batch size t is very small. In Section 6, we improve upon this algorithm to arrive at Algorithm S2. Analytic results for Algorithm S2 are provided in Section 7. A heuristic capable of significantly speeding up Algorithms S1 and S2 is presented in Section 8. In Section 9, we list our experimental results, and compare the performances of the three algorithms N, S1 and S2. We also study the performances of three faster variants N$'$, S1$'$ and S2$'$ of these algorithms. Although we have concentrated only upon the curves over prime fields, supplied in the NIST standard [9], our algorithms readily apply to other curves with cofactor 1. As mentioned in [1], cofactor values larger than 1 can be

easily handled by appending only a few bits of extra information to standard ECDSA signatures. The concluding Section 10 highlights some future research directions.

2 Notations

The elliptic-curve digital signature algorithm (ECDSA) is based upon some parameters common to all entities participating in a network.

$q =$ Order of the prime field $\mathbb{F}_q$.
$E =$ An elliptic curve $y^2 = x^3 + ax + b$ defined over the prime field $\mathbb{F}_q$.
$P =$ A random non-zero base point in $E(\mathbb{F}_q)$.
$n =$ The order of P, typically a prime.
$h =$ The cofactor $\dfrac{|E(\mathbb{F}_q)|}{n}$.

For the time being, we assume that $h = 1$, that is, $E(\mathbb{F}_q)$ is a cyclic group, and P is a generator of $E(\mathbb{F}_q)$. This is indeed the case for certain elliptic curves standardized by NIST. By Hasse's theorem, we have $|n - q - 1| \leqslant 2\sqrt{q}$. If $n \geqslant q$, an element of $\mathbb{Z}_n$ has a unique representation in $\mathbb{Z}_q$. On the other hand, if $n < q$, an element of $\mathbb{Z}_n$ has at most two representations in $\mathbb{Z}_q$. The density of elements of $\mathbb{Z}_n$ having two representations in $\mathbb{Z}_q$ is $\leqslant 2/\sqrt{q}$ which is close to zero for large values of q.

In an ECDSA signature (M, r, s), the values r and s are known modulo n. However, r corresponds to an elliptic-curve point and should be known modulo q. If r corresponds to a random point on E, it uniquely identifies an element of $\mathbb{F}_q$ with probability close to 1. In view of this, we will ignore the effect of issues associated with the ambiguous representation stated above, in the rest of this article.

Note that the ambiguities arising out of $h > 1$ and/or $q > n$ can be practically solved by appending only a few extra bits to standard ECDSA signatures [1, 3]. Consequently, our assumptions are neither too restrictive nor too impractical.

An ECDSA key pair consists of the public key Q and the private key d satisfying $Q = dP$. The steps for generating the ECDSA signature (r, s) on a message M follow.

1. $k =$ A randomly chosen element in the range $[1, n-1]$ (the session key).
2. $R = kP$.
3. $r = x(R)$ (the x-coordinate of R) reduced modulo n.
4. $s = k^{-1}(H(M) + dr) \pmod{n}$ (where H is a cryptographic hash function like SHA-1 [10]).

The following steps verify the ECDSA signature (r, s) on a message M.

1. $w = s^{-1} \pmod{n}$.
2. $u = H(M)w \pmod{n}$.
3. $v = rw \pmod{n}$.
4. $R = uP + vQ \in E(\mathbb{F}_q)$. (1)
5. Accept the signature if and only if $x(R) = r \pmod{n}$.

3 Naive Batch Verification Algorithms N and N′ for ECDSA

Throughout the rest of this paper, we plan to simultaneously verify t ECDSA signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ on messages $M_1, M_2, \ldots, M_t$. By m, we will denote 2^t.

For t signed messages (M_i, r_i, s_i), $i = 1, 2, \ldots, t$, we have

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i\right) P + \sum_{i=1}^{t} v_i Q_i. \tag{2}$$

If all the signatures belong to the same signer, we have $Q_1 = Q_2 = \cdots = Q_t = Q$ (say), and the last equation simplifies to:

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i\right) P + \left(\sum_{i=1}^{t} v_i\right) Q. \tag{3}$$

The basic idea is to compute the two sides of Eqn (2) or Eqn (3), and check for the equality. Use of these equations reduces the number of scalar multiplications from $2t$ to $[2, t+1]$, where 2 corresponds to the case where all the signatures belong to same signer, and $t+1$ corresponds to the case where the t signers are distinct from one another. However, only the x-coordinates of R_i are known from the signatures. In general, there are two y-coordinates corresponding to a given x-coordinate, but computing these y-coordinates requires taking square roots modulo q, a time-consuming operation. Moreover, there is nothing immediately available in the signatures to remove the ambiguity in these two values of y. Finally, computing all R_i using Eqn (1) misses the basic idea of batch verification, since after this expensive computation, there is only an insignificant amount of effort left to complete individual verifications of the t signatures.

ECDSA* [1], a modification of ECDSA where the entire point R is included in the signature instead of r, adapts readily to the above batch-verification idea. Nonetheless, a naive algorithm (to be denoted as Algorithm N) for the batch verification of original ECDSA signatures can be conceived of. For each i, we compute the square roots y_i of $r_i^3 + ar_i + b$ modulo q. There are (usually) 2^t choices of the square roots y_i for all $i = 1, 2, \ldots, t$. If any of these combinations of square roots satisfies Eqn (2), we accept the batch of signatures. This is definitely an obvious way of solving the ECDSA batch-verification problem, but we have not found any previous mention of this algorithm in the literature. Modular square-root computation turns out to be a costly operation. Moreover, we need to check (at most) $m = 2^t$ possible conditions for batch verification—a step that is also quite costly unless t is small.

Using a single extra bit of information in an ECDSA signature, one can unambiguously identify the *correct* square root of $r_i^3 + ar_i + b$, and thereby avoid the $\Theta(2^t)$ overhead associated with Algorithm N. This updated (and efficient) version of the naive algorithm will henceforth be denoted by Algorithm N′. Despite this updating, there is apparently nothing present in ECDSA signatures, that provides a support for quickly *computing* the correct square root. The basic aim of this paper is to develop algorithms to reduce the overhead associated with square-root calculations. In effect, we are converting ECDSA signatures to ECDSA* signatures. In that sense, this paper is not competing with but complementary to the earlier works [1, 3] on ECDSA*.

4 A New Batch-verification Algorithm for ECDSA (Algorithm S1)

In this section, we present a new algorithm to convert Eqn (2) or Eqn (3) to a form which eliminates the problems associated with the lack of knowledge of the y-coordinates of R_i. We compute the right side of Eqn (2) or Eqn (3) as efficiently as possible. The left side is not computed explicitly, but symbolically in the unknown values $y_1, y_2, \ldots, y_t$ (the y-coordinates of $R_1, R_2, \ldots, R_t$). By solving a system of linear equations over $\mathbb{F}_q$, we obtain enough information to verify the t signatures simultaneously. This new algorithm, called Algorithm S1, turns out to be faster than Algorithm N for small batch sizes (typically for $t \leqslant 4$) and for large underlying fields.

4.1 Symbolic Computation of $R = \sum_{i=1}^{t} R_i$

Let $R_i = (x_i, y_i)$. The x-coordinates $x_i = x(R_i)$ are available from the signatures, namely, $x_i = r_i$ or $x_i = r_i + n$. The second case pertains to the condition $n < q$ and has a very low probability. So we plan to ignore this case, and take $x_i = x(R_i) = r_i$. It is indeed easy to detect when the reduced x-coordinate r_i has two representatives in $\mathbb{F}_q$, and if so, we repeat Algorithm S1 for both these values.

Although the y-coordinate $y_i = y(R_i)$ is unknown to us, we know the values of

$$y_i^2 = r_i^3 + ar_i + b \pmod{q} \tag{4}$$

for all $i = 1, 2, \ldots, t$, since $R_i = (r_i, y_i)$ is a point on the curve E.

Applying the elliptic-curve point-addition formula repeatedly gives the following representation of the point $R = \sum_{i=1}^{t} R_i$:

$$R = \left(\frac{g_x(y_1, y_2, \ldots, y_t)}{h_x(y_1, y_2, \ldots, y_t)}, \frac{g_y(y_1, y_2, \ldots, y_t)}{h_y(y_1, y_2, \ldots, y_t)} \right), \tag{5}$$

where g_x, g_y, h_x, h_y are polynomials in $\mathbb{F}_q[y_1, y_2, \ldots, y_t]$. In view of Eqn (4), we may assume that these polynomials have y_i-degrees $\leqslant 1$ for all $i = 1, 2, \ldots, t$. This implies that the denominator $h_x(y)$ is of the form $u(y_2, y_3, \ldots, y_t)y_1 + v(y_2, y_3, \ldots, y_t)$. Multiplying both g_x and h_x by $u(y_2, y_3, \ldots, y_t)y_1 - v(y_2, y_3, \ldots, y_t)$ and using Eqn (4), we can eliminate y_1 from the denominator. Repeating this successively for $y_2, y_3, \ldots, y_t$ allows us to represent the point R as a pair of polynomial expressions:

$$R = (R_x(y_1, y_2, \ldots, y_t), R_y(y_1, y_2, \ldots, y_t)) \tag{6}$$

with the polynomials R_x and R_y linear individually with respect to all y_i. It is useful to clear the denominator after every symbolic addition instead of only once after the entire sum $R = \sum_{i=1}^{t} R_i$ is computed symbolically. It is easy to establish that R_x is a polynomial with each non-zero term having even total degree, whereas R_y is a polynomial with each non-zero term having odd total degree (See Appendix A).

From the right side of Eqn (2) or Eqn (3), we compute the x- and y-coordinates of R as $R = (\alpha, \beta)$ for some $\alpha, \beta \in \mathbb{F}_q$. This gives us two initial multivariate equations:

$$R_x(y_1, y_2, \ldots, y_t) = \alpha, \tag{7}$$

$$R_y(y_1, y_2, \ldots, y_t) = \beta. \tag{8}$$

4.2 Solving the Multivariate Equations

We treat Eqns (7) and (8) as linear equations in the square-free monomials y_i, y_iy_j, $y_iy_jy_k$, and so on. R_x contains non-zero terms involving only the even-degree monomials, that is, y_iy_j, $y_iy_jy_ky_l$, and so on. There are exactly $\mu = 2^{t-1} - 1 = \frac{m}{2} - 1$ such monomials, where $m = 2^t$. We name these monomials as $z_1, z_2, \ldots, z_\mu$, and take out the constant term from R_x to rewrite Eqn (7) as

$$\rho_{1,1}z_1 + \rho_{1,2}z_2 + \cdots + \rho_{1,\mu}z_\mu = \alpha_1. \tag{9}$$

If we square both sides of this equation, and use Eqn (4) to eliminate all squares of variables, we obtain another linear equation:

$$\rho_{2,1}z_1 + \rho_{2,2}z_2 + \cdots + \rho_{2,\mu}z_\mu = \alpha_2. \tag{10}$$

By repeated squaring, we generate a total of μ linear equations in $z_1, z_2, \ldots, z_\mu$. We then solve the resulting system and obtain the values of $z_1, z_2, \ldots, z_\mu$.

If the system is not of full rank, we make use of Eqn (8) as follows. Each non-zero term in R_y has odd degree. However, the equation $R_y^2 = \beta^2$ (along with the substitution given by Eqn (4)) leads to a linear equation in the even-degree monomials $z_1, z_2, \ldots, z_\mu$ only. Repeated squaring of this equation continues to generate a second sequence of linear equations in $z_1, z_2, \ldots, z_\mu$.

We expect to obtain μ linearly independent equations from these two sequences.

4.3 A Strategy for Faster Equation Generation

There are indeed other ways of generating new linear equations in $z_1, z_2, \ldots, z_\mu$. Let

$$\rho_1z_1 + \rho_2z_2 + \cdots + \rho_\mu z_\mu = \gamma \tag{11}$$

be an equation already generated, and let $f(z_1, z_2, \ldots, z_\mu)$ be any $\mathbb{F}_q$-linear combination of the monomials $z_1, z_2, \ldots, z_\mu$. Simplification of the equation

$$(\rho_1z_1 + \rho_2z_2 + \cdots + \rho_\mu z_\mu)f(z_1, z_2, \ldots, z_\mu) = \gamma f(z_1, z_2, \ldots, z_\mu)$$

using Eqn (4) again yields a linear equation in $z_1, z_2, \ldots, z_\mu$. In particular, the choice $f(z_1, z_2, \ldots, z_\mu) = z_i$ with a small degree of z_i typically leads to a faster generation of a new equation than squaring Eqn (11). Our experiments indicate that we can generate a full-rank system by monomial multiplications and a few squaring operations. Moreover, only Eqn (7) suffices to generate a uniquely solvable linearized system.

4.4 Retrieving the Unknown y-coordinates

The final step in Algorithm S1 involves the determination of the y-coordinates y_i of the points R_i. Multiplying both sides of Eqn (8) by y_1 gives an equation of the form

$$\beta y_1 = \epsilon_0 + \epsilon_1z_1 + \epsilon_2z_2 + \cdots + \epsilon_\mu z_\mu.$$

Substitution of the values of z_i available from the previous stage gives y_1 (provided that $\beta \neq 0$). Subsequently, the values y_i for $i = 2, 3, \ldots, t$ can be obtained by dividing

the known value of $y_1 y_j$ by y_1 provided that $y_1 \neq 0$. Even if $y_1 = 0$, we can multiply Eqn (8) by y_2 to solve for y_2. If $y_2 \neq 0$, we are allowed to compute $y_i = (y_2 y_i)/y_2$ for $i \geqslant 3$. If $y_2 = 0$ too, we compute y_3 by directly using Eqn (8), and so on. The only condition that is necessary to solve for all y_i values uniquely is $\beta \neq 0$, where β is the y-coordinate of the point on the right side of Eqn (2) (or Eqn (3)).

We finally check whether Eqn (4) is valid for all $i = 1, 2, \ldots, t$. If so, all the signatures are verified simultaneously. If one or more of these equations fail(s) to hold, batch verification fails.

In short, Algorithm S1 uniquely reconstructs the points R_i with $x(R_i) = r_i$. The computations do not involve taking modular square roots in $\mathbb{F}_q$. We also avoid computing the points $R'_i = u_i P + v_i Q_i$ needed in individual verification. The final check ($y_i^2 = r_i^3 + a r_i + b$) guarantees that the reconstructed points really lie on the curve. In the next section, we prove that the reconstruction process succeeds with very high probability. Moreover, for small batch sizes, the reconstruction process is efficient.

5 Analysis of Algorithm S1

5.1 Running Time

The count of monomials handled during the equation-generation and equation-solving stages is $\mu = 2^{t-1} - 1 = \frac{m}{2} - 1$ which grows exponentially with t. Determination of the Eqns (7) and (8) needs $t - 1$ symbolic additions involving rational functions with at most $\Theta(m)$ non-zero terms. Each symbolic addition is followed by at most t uses of Eqn (4). Therefore, the symbolic derivation of R requires $\mathrm{O}(mt^2)$ operations in $\mathbb{F}_q$. The subsequent generation of the $\mu \times \mu$ linearized system requires $\mathrm{O}(m^2 t)$ field operations. Finally, Gaussian elimination on an $\mu \times \mu$ system demands $\Theta(m^3)$ field operations. Retrieving individual y_i values calls for $\mathrm{O}(mt^2)$ (usually $\mathrm{O}(mt)$) field operations. The running time of Algorithm S1 is dominated by the linear system-solving stage. Evidently, Algorithm S1 becomes impractical except only for small values of t.

It is worthwhile to investigate the running time of the naive Algorithm N. First, this algorithm needs to compute t modular square roots in the field $\mathbb{F}_q$. Each such square-root computation (for example, by the Tonelli-Shanks algorithm [13]) involves an exponentiation in $\mathbb{F}_q$. Subsequently, one needs to check at most $m = 2^t = 2(\mu + 1)$ conditions, with each check involving the computation of the sum of t points on the curve. Therefore, the total running time of Algorithm N is $\mathrm{O}((\sigma + m)t)$, where σ is the time for computing one square root in $\mathbb{F}_q$. Thus, Algorithm S1 outperforms Algorithm N only in situations where σ is rather large compared to m. This happens typically when the batch size t is small and the field size q is large.

5.2 Unique Solvability of the Linearized System

In Algorithm S1, we solve a linearized $\mu \times \mu$ system to obtain the values of the even-degree monomials $z_1, z_2, \ldots, z_\mu$ in the unknown y-coordinates $y_1, y_2, \ldots, y_t$. Let us call the coefficient matrix M. In order that the linearized system is uniquely solvable, we require $\det M \neq 0$. We now investigate how often this condition is satisfied, and also how we can force this condition to hold in most cases.

For a moment, let us treat the x-coordinates $r_1, r_2, \ldots, r_t$ as symbols. But then the failure condition $\det M = 0$ can be rephrased in terms of a multivariate polynomial equation in $r_1, r_2, \ldots, r_t$. Let us denote this equation as $D(r_1, r_2, \ldots, r_t) = 0$. If D is identically zero, then any values of $r_1, r_2, \ldots, r_t$ constitute a root of D. We explain shortly how this situation can be avoided.

Assume that D is not identically zero. Let δ be the maximum degree of each individual r_i in D. One can derive that $\delta \leqslant \left(2^{2t+3\lceil \log_2 t \rceil + 2} + 3\right)\left(2^{2^{t-1}-1} - 1\right) \approx 2^{2^{t-1}+2t+3\lceil \log_2 t \rceil + 1}$ (See Appendix B). If we restrict our attention to the values $t \leqslant 6$, we have $\delta \leqslant 2^{54}$. The maximum number of roots of D is bounded below $t\delta q^{t-1}$ (See Appendix C). The total number of t-tuples $(r_1, r_2, \ldots, r_t)$ over $\mathbb{F}_q$ is q^t. Therefore, a randomly chosen tuple $(r_1, r_2, \ldots, r_t)$ is a root of D with probability $\leqslant t\delta q^{t-1}/q^t = t\delta/q$. If we use the inequalities $t \leqslant 6$, $\delta \leqslant 2^{54}$ and $q \geqslant 2^{160}$, we conclude that this probability is less than 2^{-103}. Therefore, if D is not the zero polynomial, we can solve for $z_1, z_2, \ldots, z_\mu$ uniquely with very high probability.

What remains is to propose a way to avoid the condition $D = 0$. We start with any t randomly chosen ECDSA signatures with r-values $r_1, r_2, \ldots, r_t$. We then choose any sequence of squaring and multiplication by z_i in order to arrive at a linear system in $z_1, z_2, \ldots, z_\mu$. If the corresponding coefficient matrix M is not invertible, we discard the chosen sequence of squaring and multiplication. This is because $\det M = 0$ implies that either D is the zero polynomial or the chosen $r_1, r_2, \ldots, r_t$ constitute a root of a non-zero D. The second case is extremely unlikely. With high probability, we, therefore, conclude that the chosen sequence of squaring and multiplication gives $D = 0$ identically. We change the sequence, and repeat the above process until we come across the situation where $r_1, r_2, \ldots, r_t$ do not constitute a root of the non-zero polynomial equation $D(r_1, r_2, \ldots, r_t) = 0$. This implies that D is not identically zero, and randomly chosen $r_1, r_2, \ldots, r_t$ satisfy $D(r_1, r_2, \ldots, r_t) = 0$ with very low probability. We keep this sequence for all future invocations of our batch-verification algorithm.

Table 1 lists some sequences of squaring and multiplication, that work for NIST prime curves. Here, S stands for a squaring step, whereas a monomial (like y_2y_4) stands for multiplication by that monomial. In all these cases, we use only Eqn (7), whereas Eqn (8) is used only for the unique determination of individual y_i values. These sequences depend upon t alone, but not on the NIST curves. For other curves, this method is expected to work equally well. Indeed, we may consider $D(r_1, r_2, \ldots, r_t)$ as a polynomial in $\mathbb{Z}[r_1, r_2, \ldots, r_t]$. If D is not identically zero, then it is identically zero modulo only a finite number of primes (the common prime divisors of the coefficients of D).

Table 1. Sequences to generate linearized systems for NIST prime curves

t	**Sequence in the linearization phase**
2	No squaring or multiplication needed
3	y_1y_2, y_1y_3
4	$y_1y_2, y_1y_3, y_1y_4, y_2y_3, y_3y_4, y_1y_4$
5	$y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_2y_3, y_2y_4, y_4y_5, y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_1y_2, y_2y_4, y_2y_3$
6	$y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_1y_6, y_2y_3, y_2y_4, y_2y_5, y_1y_2, y_3y_4, y_3y_5, y_1y_5, y_1y_6, y_1y_2y_3y_6, y_1y_5,$ $y_1y_4, y_1y_3, y_1y_2y_3y_6, y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_2y_5, y_2y_3, S, y_2y_6, y_4y_6, y_3y_6, y_5y_6, y_1y_5$

5.3 Security Analysis

In Algorithm S1, we reconstruct the points R_i with x-coordinates $x(R_i) = r_i$ by forcing the condition $R = \sum_{i=1}^{t} R_i = \sum_{i=1}^{t} R'_i = R'$, where $R'_i = u_i P + v_i Q_i$. Suppose that an adversary too can force the condition $R = R'$. The adversary must also reveal the x-coordinates $r_1, r_2, \ldots, r_t$ as parts of ECDSA signatures. Given these x-coordinates and the condition $R = R'$, there exists (with high probability) a unique solution for the corresponding y-coordinates $y_1, y_2, \ldots, y_t$ of $R_1, R_2, \ldots, R_t$. This solution can be computed by the adversary, for example, using Algorithm S1 (or by taking modular square roots in $\mathbb{F}_q$ as in Algorithm N). So long as t is restricted to small constant values (like $t \leqslant 6$), the adversary requires only moderate computing resources for determining $y_1, y_2, \ldots, y_t$ uniquely. This implies that although the adversary needs to reveal only the x-coordinates r_i, (s)he essentially *knows* the full points R_i. But these points $R_1, R_2, \ldots, R_t$ satisfy the standard batch-verification condition for ECDSA*. That is, if the adversary can fool Algorithm S1, (s)he can fool the standard ECDSA* batch-verification algorithm too. It follows that Algorithm S1 is no less secure than the standard batch-verification algorithm for ECDSA*. Conversely, if an adversary can fool any ECDSA* batch-verification algorithm, (s)he can always fool any ECDSA batch-verification algorithm, since ECDSA signatures are only parts of ECDSA* signatures. To sum up, Algorithm S1 is as secure as standard ECDSA* batch verification [7].

An analysis of the security of Algorithm N is also worth including here. Suppose that an adversary can pass one of the $m = 2^t$ checks in Algorithm N along with disclosing $r_1, r_2, \ldots, r_t$. The correct choices y_i of the square roots of $r_i^3 + ar_i + b$ (that is, those choices corresponding to the successful check) constitute a case of fulfillment of the ECDSA* batch-verification criterion. Consequently, Algorithm N too is as secure as standard ECDSA* batch verification.

5.4 Cases of Failure for Algorithm S1

Our Monte Carlo batch-verification Algorithm S1 may fail for a few reasons. We now argue that these cases of failure are probabilistically very rare.

1. Taking $x_i = r_i$ blindly is a possible cause of failure for Algorithm S1. As discussed earlier, this situation has a very low probability. Furthermore, it is easy to identify when this situation occurs. In case of ambiguity in the values of x_i, we can repeat Algorithm S1 for all possible candidate tuples $(x_1, x_2, \ldots, x_t)$. If the points R_i are randomly chosen in $E(\mathbb{F}_q)$, most of these x_i values are unambiguously available to us, and there should not be many repeated runs (if any) of Algorithm S1. Repeated runs, if necessary, may be avoided, because doing so goes against the expected benefits achievable by batch verification.
2. Although we are able to identify good sequences of squaring and multiplication in order to force the determinant polynomial $D(r_1, r_2, \ldots, r_t)$ to be not identically zero, roots of this polynomial may appear in some cases of ECDSA signatures. We have seen that if $r_1, r_2, \ldots, r_t$ are randomly chosen, the probability of this situation is no more than 2^{-103} (for $t \leqslant 6$).

3. Eqn (5) is derived using the point-addition formula on the curve E, which is different from the doubling formula. So long as we work symbolically using the unknown quantities $y_1, y_2, \ldots, y_t$, it is impossible to predict when the two points being added turn out to be equal. If $R_1, R_2, \ldots, R_t$ are randomly chosen from $E(\mathbb{F}_q)$, the probability of this occurrence is extremely low.
4. Algorithm S1 fails if R' is the point at infinity or lies on the x-axis ($\beta = 0$). In that case, one should resort to individual verification. For randomly chosen session keys, this case occurs with a very small probability (nearly $4/q$).

6 A More Efficient Batch-verification Algorithm (Algorithm S2)

The linearization stage in Algorithm S1 (requiring $\mathrm{O}(m^2t)$ field operations) and the subsequent Gaussian-elimination stage ($\mathrm{O}(m^3)$ field operations) are rather costly, m being an exponential function of the batch size t. Our second symbolic-manipulation algorithm S2 avoids these two stages altogether.

Algorithm S1 uniquely solves for the monomials $z_1, z_2, \ldots, z_\mu$ using the equation $R_x = \alpha$ only. At this point, there are only two possible solutions for the y_i values: $(y_1, y_2, \ldots, y_t)$ and $(-y_1, -y_2, \ldots, -y_t)$. This *sign* ambiguity is eliminated by using the other equation $R_y = \beta$. As mentioned in connection with the security analysis of Algorithm N, the exact determination of these signs is not important. In other words, we would be happy even if we can determine each y_i correctly up to multiplication by ± 1. This, in turn, implies that if we have any multivariate equation (linear in y_i) of the form $uy_i + v = 0$ (where u, v are polynomials in $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_t$), we do not mind multiplying this equation by $uy_i - v$ so that $\pm y_i$ satisfy $u^2y_i^2 - v^2 = 0$. But $y_i^2 = r_i^3 + ar_i + b$, so we have $u_i^2(r_i^3 + ar_i + b) - v_i^2 = 0$, an equation in which y_i is *eliminated*. This observation leads to Algorithm S2.

Like Algorithm S1, we first symbolically compute $R = \sum_{i=1}^{t} R_i$, and arrive at Eqns (7) and (8). Then, we consider only the multivariate equation $R_x - \alpha = 0$ linear individually in each y_i. We first eliminate y_1, and with substitutions given by Eqn (4) for $i = 2, 3, \ldots, t$, we arrive at a multivariate equation in $y_2, y_3, \ldots, y_t$, again linear in each of these variables. We eliminate y_2 from this equation, and arrive at a multivariate equation in $y_3, y_4, \ldots, y_t$. We repeat this process until all variables $y_1, y_2, \ldots, y_t$ are eliminated. If the polynomial after all these eliminations reduces to zero, the original equation $R_x = \alpha$ is consistent with respect to $y_i^2 = r_i^3 + ar_i + b$ for all $i = 1, 2, \ldots, t$.

We may likewise eliminate $y_1, y_2, \ldots, y_t$ from $R_y - \beta = 0$ too, but this is not necessary, because it suffices to know y_i uniquely up to multiplication by ± 1.

Some comments on efficient implementations of the elimination stage are now in order. First, we are not using Eqn (8) at all in Algorithm S2. Consequently, it is not necessary to compute the polynomial R_y. However, in the symbolic-computation stage, we need to compute all intermediate y-coordinates, since they are needed in the final value of R_x. The computation of only the last y-coordinate R_y may be avoided. Still, this saves quite some amount of effort ($\mathrm{O}(mt)$ field operations, to be precise). This saving does not affect the theoretical complexity of the algorithm in the big-Oh notation, but its practical effects are noticeable.

The second issue is that the polynomials u and v in each elimination step have some nice properties. Throughout this step, $\phi = uy_i + v$ and v are polynomials with each

non-zero term having even degree, whereas u is a polynomial with each non-zero term having odd degree. In particular, when the first $t-2$ y-coordinates are eliminated, we have $\phi = uy_{t-1}y_t + v$ with $u, v \in \mathbb{F}_q$. Elimination of y_{t-1} eliminates y_t too, so an explicit elimination of y_t is not necessary.

The y-coordinates $y_1, y_2, \ldots, y_t$ are not explicitly reconstructed in Algorithm S2. However, if necessary, we can compute two sets of solutions $y_1, y_2, \ldots, y_t$ and $-y_1$, $-y_2, \ldots, -y_t$ by using the values of $\phi = uy_i + v$ for $i = t-1, t-2, \ldots, 2, 1$. The sign ambiguity can be removed by using $R_y = \beta$. Algorithm S2 does not include this reconstruction phase, since this is cryptographically unimportant. However, we use this result in the security proof for S2.

It is also important to note that the determination of individual y_i values is cryptographically unimportant for Algorithm S1 too, since $R_x = \alpha$ already identifies exactly two solutions for the reconstructed points. If these steps are omitted, the batch-acceptance criterion would match z_i^2 against appropriate products of $r_j^3 + ar_j + b$ for all $i = 1, 2, \ldots, \mu$. In fact, it suffices to consider only the monomials z_i of degree 2. However, the unique determination of y_i values takes only an insignificant fraction of time in Algorithm S1, so it does not practically matter to make a choice between whether we carry out these steps or not.

7 Analysis of Algorithm S2

7.1 Running Time

The symbolic computation of (R_x, R_y) involves $\mathrm{O}(mt^2)$ field operations (as in Algorithm S1). Subsequently, we start with the polynomial $\phi = R_x - \alpha$ with at most $\mu + 1 = \frac{m}{2} + 1$ non-zero terms. Elimination of y_i requires computing the squares u^2 and v^2, carrying out the polynomial arithmetic $u^2(r_i^3 + ar_i + b) - v^2$, and $t - i$ substitutions of y_j^2 by $r_j^3 + ar_j + b$. Therefore, the reduction of ϕ too requires $\mathrm{O}(mt^2)$ field operations. This is significantly better than the $\mathrm{O}(m^3)$ operations needed by Algorithm S1. Moreover, Algorithm S2 outperforms Algorithm N for a wide range of t and q, since the condition $(\sigma + m)t \gg mt^2$ is more often satisfied than the condition $(\sigma + m)t \gg m^3$.

7.2 Security Analysis

We establish the equivalence between the security of Algorithm S2 and the security of standard ECDSA* batch verification, as we have done for the earlier algorithms (N and S1). Suppose that an adversary reveals the x-coordinates $r_1, r_2, \ldots, r_t$ in ECDSA signatures which pass the batch-verification procedure of Algorithm S2. We mentioned above that there are exactly two solutions $(y_1, y_2, \ldots, y_t)$ and $(-y_1, -y_2, \ldots, -y_t)$ consistent with $R_x - \alpha = 0$ and $y_i^2 = r_i^3 + ar_i + b$ for $i = 1, 2, \ldots, t$. One of these solutions corresponds to the ECDSA* signatures based upon the disclosed values $r_1, r_2, \ldots, r_t$. It is that solution that would pass $R_y = \beta$. To sum up, the adversary can forge the standard ECDSA* batch-verification algorithm. Moreover, this forging procedure which essentially involves the unique reconstruction of the points $R_i = (r_i, y_i)$ is practical for any adversary with only a moderate amount of computing resources, so long as t is restricted only to small values (the only cases where we can apply S2).

8 Efficient Variants of S1 and S2

In Algorithm S1, we generate a system of linearized equations in $\frac{m}{2} - 1 = 2^{t-1} - 1$ monomials. Solving the resulting equation turns out to be the costliest step of Algorithm S1, demanding $\Theta(m^3)$ field operations. In Algorithm S2, the symbolic computation of $R = (R_x, R_y)$ turns out to be the most time-consuming step. This step calls for $\Theta(mt^2)$ field operations. The elimination phase too calls for $\Theta(mt^2)$ operations.

In this section, we explain a strategy to reduce the number of monomials in Algorithms S1 and S2. So far, we have been symbolically computing the point $R = \sum_{i=1}^{t} R_i$, and equating the symbolic sum to $R' = (\alpha, \beta)$. This results in polynomial expressions with $\Theta(2^{t-1})$ (that is, $\Theta(m)$) non-zero terms.

Now, let $\tau = \lceil t/2 \rceil$. We symbolically compute the two sums:

$$R^{(1)} = \sum_{i=1}^{\tau} R_i \text{ and } R^{(2)} = R' - \sum_{i=\tau+1}^{t} R_i. \tag{12}$$

The polynomial expressions involved in $R^{(1)}$ and $R^{(2)}$ contain only $\Theta(2^{\tau})$, that is, $\Theta(\sqrt{m})$ non-zero terms. So computing these two symbolic sums needs $\Theta(2^{\tau}\tau^2)$, that is, $\Theta(\sqrt{m}t^2)$ field operations which is significantly smaller than the $\Theta(mt^2)$ operations associated with the symbolic computation of the complete sum $\sum_{i=1}^{t} R_i$. The condition $R = R'$ is equivalent to the condition $R^{(1)} = R^{(2)}$. Using this new condition helps us in speeding up the subsequent steps too.

8.1 Algorithm S1′

The symbolic computation of R in Algorithm S1 can be replaced by the two symbolic computations given by Eqn (12). In that case, we replace the initial equations $R_x = \alpha$ and $R_y = \beta$ by the two equations $x(R^{(1)}) = x(R^{(2)})$ and $y(R^{(1)}) = y(R^{(2)})$. It is easy to argue that $x(R^{(1)})$ is a polynomial in $y_1, y_2, \ldots, y_\tau$ with each non-zero term having even degree, whereas $y(R^{(1)})$ is a polynomial in $y_1, y_2, \ldots, y_\tau$ with each non-zero term having odd degree. That is, the number of non-zero terms in these two expressions is $2^{\tau-1} = \frac{\sqrt{m}}{2}$. However, the presence of $R' = (\alpha, \beta)$ on the right side of the expression for $R^{(2)}$ (Eqn 12) lets both $x(R^{(2)})$ and $y(R^{(2)})$ contain all (square-free) monomials in $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$ (both even and odd degrees). There are exactly $2^{\lfloor t/2 \rfloor} - 1 \leqslant \sqrt{m} - 1$ monomials in these two expressions. In the linearized system that we subsequently generate, we consider, as variables, only the even-degree monomials in $y_1, y_2, \ldots, y_\tau$ and all monomials in $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$.

We start with the equation $x(R^{(1)}) = x(R^{(2)})$. Subsequently, we keep on squaring the equation $x(R^{(1)}) = x(R^{(2)})$ (and substituting values of y_i^2 wherever necessary). This sequence does not increase the number of monomials in the linearized equations. More precisely, for any $j \geqslant 0$, the equation $x(R^{(1)})^{2^j} = x(R^{(2)})^{2^j}$ contains only the $\Theta(\sqrt{m})$ monomials with which we start. If we fail to obtain a linearized system of full rank, we start squaring the other initial equation $y(R^{(1)}) = y(R^{(2)})$. For any $j \geqslant 1$, the equation $y(R^{(1)})^{2^j} = y(R^{(2)})^{2^j}$ again contains only the monomials with which we start. In all the cases studied, we have been able to obtain a full-rank linearized

system by squaring the two initial equations. Since the number of linearized variables is $\Theta(\sqrt{m})$, the linearization step of Algorithm S1 now reduces to $\mathrm{O}(mt)$ field operations. Finally, we solve a system with $\Theta(\sqrt{m})$ variables using $\Theta(m^{3/2})$ field operations.

To sum up, using the trick introduced in this section decreases the number of field operations from $\Theta(m^3)$ to $\Theta(m^{3/2})$. Let us plan to call this efficient variant of S1 as S1′. Fundamentally, S1′ is not a different algorithm from S1. In particular, the security of S1′ is the same as the security of S1 (in fact, little better, because fewer linearized equations are involved). However, the reduction in the running time is very significant, both theoretically and practically.

8.2 Algorithm S2′

Instead of starting with $\phi = R_x - \alpha$, Algorithm S2′ starts with the initial expression

$$\phi = x(R^{(1)}) - x(R^{(2)}). \tag{13}$$

We then repeatedly eliminate $y_1, y_2, \ldots, y_t$. Although the initial expression of ϕ contains much less number of monomials than in the original Algorithm S2, elimination of y_1 itself introduces many new monomials in ϕ, that is, soon ϕ becomes almost *full*. Consequently, the elimination phase continues to make $\Theta(mt^2)$ field operations as before, that is, the theoretical running time of S2′ is the same as that of S2. Still, the effects of our heuristic are clearly noticeable in practical implementations.

As described in Section 6, the y-coordinates $y(R^{(1)})$ and $y(R^{(2)})$ need not be computed. It is, however, necessary to symbolically compute the y-coordinates of all intermediate sums.

9 Experimental Results

Our batch-verification algorithms are implemented using the GP/PARI calculator [11] (version 2.3.5). Our choice of this implementation platform is dictated by the symbolic-computation facilities and an easy user interface provided by the calculator. All experiments are carried out in a 2.33 MHz Xeon server running Mandriva Linux Version 2010.1. The GNU C compiler 4.4.3 is used for compiling the GP/PARI calculator.

In Table 2, we list the average times for carrying out single scalar multiplications in the NIST prime curves. This table also lists the times for single square-root calculations in the underlying fields. Table 3 lists the overheads associated with the three algorithms N, S1 and S2, and their variants N′, S1′ and S2′. These overhead figures do not include the scalar-multiplication times. The algorithms S1, S1′ and S2 become impractical for batch sizes $t > 6$, so these algorithms are not implemented for $t = 7$ and $t = 8$.

Table 2. Timings (ms) for NIST prime curves

	P-192	P-224	P-256	P-384	P-521
Time for Scalar Multiplication (in $E(\mathbb{F}_q)$)	1.82	2.50	3.14	7.33	14.38
Time for Square-root (in $\mathbb{F}_q$)	0.06	0.35	0.09	0.26	0.67

Table 3. Overheads (ms) for different batch-verification algorithms

	Naive (N)							Naive (N′)						
	t							t						
Curve	2	3	4	5	6	7	8	2	3	4	5	6	7	8
P-192	0.18	0.39	0.76	1.57	3.40	7.71	17.00	0.13	0.19	0.26	0.33	0.39	0.46	0.52
P-224	0.81	1.34	2.04	3.29	5.63	10.60	21.50	0.71	1.06	1.42	1.78	2.14	2.49	2.85
P-256	0.24	0.49	0.97	1.95	4.18	9.27	20.85	0.19	0.29	0.38	0.48	0.58	0.68	0.78
P-384	0.66	1.15	1.95	3.51	6.76	13.80	29.90	0.53	0.81	1.08	1.35	1.62	1.90	2.17
P-521	1.66	2.70	4.21	6.73	11.63	21.00	43.10	1.36	2.05	2.74	3.42	4.11	4.80	5.49

	Symbolic (S1)					Symbolic (S1′)				
	t					t				
Curve	2	3	4	5	6	2	3	4	5	6
P-192	0.14	0.57	2.01	8.66	40.50	0.07	0.20	0.70	1.60	4.40
P-224	0.15	0.60	2.10	9.50	45.60	0.07	0.20	0.80	1.80	4.70
P-256	0.16	0.61	2.17	9.78	46.30	0.08	0.21	0.82	1.90	4.90
P-384	0.18	0.74	2.71	12.56	62.10	0.08	0.30	0.90	2.20	6.10
P-521	0.22	0.90	3.45	16.80	88.40	0.12	0.40	1.30	2.90	8.00

	Symbolic (S2)					Symbolic (S2′)						
	t					t						
Curve	2	3	4	5	6	2	3	4	5	6	7	8
P-192	0.07	0.30	0.76	2.39	6.65	0.07	0.11	0.32	0.61	1.14	2.36	5.46
P-224	0.07	0.32	0.84	2.53	7.11	0.07	0.12	0.33	0.64	1.21	2.51	5.91
P-256	0.08	0.32	0.80	2.51	7.08	0.08	0.12	0.33	0.64	1.22	2.52	5.88
P-384	0.09	0.37	0.91	2.85	8.15	0.09	0.14	0.38	0.72	1.41	2.95	7.12
P-521	0.11	0.44	1.07	3.45	10.02	0.11	0.18	0.42	0.95	1.76	3.72	9.26

Table 4 records the speedup values achieved by the six algorithms N, N′, S1, S1′, S2 and S2′. Here, the speedup is computed with respect to individual verification, and incorporates both scalar-multiplication times and batch-verification overheads. The maximum achievable speedup values (t in the case of same signer, and $2t/(t+1)$ in the case of different signers) are also listed in Table 4, to indicate how our batch-verification algorithms compare with the ideal cases. The maximum speedup obtained by our fully ECDSA-compliant algorithms is 6.20 in the case of same signer, and 1.70 in the case of different signers, both achieved by Algorithm S2′ for the curve P-521 and for $t = 7$.

From Table 4, it is evident that one should use Algorithm S2′ if extra information (a bit identifying the correct square root of each $r_i^3 + ar_i + b$) is not available. In this case, the optimal batch size is $t = 7$ (or $t = 6$ if the underlying field is small). If, on the other hand, disambiguating extra bits are appended to ECDSA signatures, one should use S2′ for $t \leqslant 4$ for (curves over) small fields and for $t \leqslant 6$ (or $t \leqslant 7$) for large fields. If the batch size increases beyond these bounds, it is preferable to use Algorithm N′.

10 Conclusion

In this paper, we have proposed six algorithms for the batch verification of ECDSA signatures. To the best of our knowledge, these are the first batch-verification algorithms

Table 4. Speedup obtained by different batch-verification algorithms

		Same signer							Different signers						
Curve	t	Ideal	N	N′	S1	S1′	S2	S2′	Ideal	N	N′	S1	S1′	S2	S2′
P-192	2	2.00	1.91	1.94	1.93	1.96	1.96	1.96	1.33	1.29	1.30	1.30	1.32	1.32	1.32
	3	3.00	2.71	2.86	2.59	2.84	2.77	2.91	1.50	1.42	1.46	1.39	1.46	1.44	1.48
	4	4.00	3.31	3.75	2.58	3.35	3.31	3.68	1.60	1.48	1.56	1.31	1.49	1.48	1.55
	5	5.00	3.49	4.62	1.48	3.47	3.02	4.28	1.67	1.46	1.62	0.93	1.45	1.37	1.58
	6	6.00	3.10	5.46	0.49	2.72	2.12	4.57	1.71	1.35	1.67	0.41	1.27	1.13	1.57
	7	7.00	2.24	6.28	–	–	–	4.25	1.75	1.14	1.70	–	–	–	1.51
	8	8.00	1.41	7.07	–	–	–	3.20	1.78	0.87	1.73	–	–	–	1.33
P-224	2	2.00	1.72	1.75	1.94	1.97	1.97	1.97	1.33	1.20	1.22	1.31	1.32	1.32	1.32
	3	3.00	2.37	2.48	2.68	2.88	2.82	2.93	1.50	1.32	1.36	1.42	1.47	1.45	1.48
	4	4.00	2.84	3.12	2.82	3.45	3.42	3.75	1.60	1.38	1.44	1.37	1.50	1.50	1.56
	5	5.00	3.02	3.70	1.72	3.68	3.32	4.43	1.67	1.37	1.49	1.02	1.49	1.43	1.60
	6	6.00	2.82	4.23	0.59	3.09	2.48	4.83	1.71	1.30	1.53	0.48	1.35	1.22	1.60
	7	7.00	2.24	4.70	–	–	–	4.66	1.75	1.14	1.56	–	–	–	1.55
	8	8.00	1.51	5.13	–	–	–	3.67	1.78	0.91	1.58	–	–	–	1.41
P-256	2	2.00	1.93	1.94	1.95	1.97	1.97	1.97	1.33	1.30	1.31	1.31	1.32	1.32	1.32
	3	3.00	2.78	2.88	2.73	2.90	2.85	2.94	1.50	1.44	1.47	1.43	1.48	1.46	1.49
	4	4.00	3.46	3.78	2.97	3.54	3.55	3.80	1.60	1.51	1.56	1.41	1.52	1.52	1.57
	5	5.00	3.82	4.67	1.96	3.84	3.57	4.54	1.67	1.51	1.63	1.10	1.51	1.47	1.61
	6	6.00	3.60	5.52	0.72	3.37	2.82	5.02	1.71	1.44	1.67	0.55	1.40	1.30	1.62
	7	7.00	2.83	6.36	–	–	–	5.00	1.75	1.28	1.71	–	–	–	1.59
	8	8.00	1.85	7.18	–	–	–	4.13	1.78	1.02	1.73	–	–	–	1.47
P-384	2	2.00	1.91	1.93	1.98	1.99	1.99	1.99	1.33	1.29	1.30	1.32	1.33	1.33	1.33
	3	3.00	2.78	2.85	2.86	2.94	2.93	2.97	1.50	1.44	1.46	1.46	1.48	1.48	1.49
	4	4.00	3.53	3.74	3.38	3.77	3.77	3.90	1.60	1.52	1.56	1.49	1.56	1.56	1.58
	5	5.00	4.03	4.59	2.69	4.35	4.19	4.77	1.67	1.54	1.62	1.30	1.59	1.57	1.64
	6	6.00	4.11	5.42	1.15	4.24	3.86	5.47	1.71	1.51	1.66	0.78	1.53	1.48	1.67
	7	7.00	3.61	6.23	–	–	–	5.83	1.75	1.42	1.70	–	–	–	1.67
	8	8.00	2.63	7.01	–	–	–	5.38	1.78	1.22	1.72	–	–	–	1.60
P-521	2	2.00	1.89	1.91	1.98	1.99	1.99	1.99	1.33	1.28	1.29	1.33	1.33	1.33	1.33
	3	3.00	2.74	2.80	2.91	2.96	2.95	2.98	1.50	1.43	1.45	1.48	1.49	1.49	1.50
	4	4.00	3.49	3.66	3.57	3.83	3.86	3.94	1.60	1.51	1.54	1.53	1.57	1.58	1.59
	5	5.00	4.05	4.48	3.16	4.54	4.46	4.84	1.67	1.55	1.60	1.40	1.61	1.60	1.65
	6	6.00	4.27	5.26	1.47	4.69	4.45	5.65	1.71	1.54	1.65	0.91	1.59	1.56	1.68
	7	7.00	4.05	6.02	–	–	–	6.20	1.75	1.48	1.68	–	–	–	1.70
	8	8.00	3.20	6.74	–	–	–	6.05	1.78	1.33	1.71	–	–	–	1.66

ever proposed for ECDSA. In particular, development of algorithms based upon symbolic manipulations appears to be a novel approach in the history of batch-verification algorithms. There are several ways to extend our study, some of which are listed below.

- Section 8 describes a way to reduce the running time of the symbolic-addition phase of Algorithm S2 from $\mathrm{O}(mt^2)$ to $\mathrm{O}(\sqrt{m}t^2)$. An analogous speedup for the elimination phase would be very useful.
- Our best symbolic-computation algorithm runs in $\mathrm{O}(mt^2)$ time. Removal of a factor of t (that is, designing an $\mathrm{O}(mt)$-time algorithm) would be useful to achieve higher speedup values.
- It is of interest to study our algorithms in conjunction with the earlier works [1, 3] on ECDSA*.
- Our batch verification algorithms can be easily ported to other curves (like the Koblitz and Pseudorandom families recommended by NIST). Solving quadratic equations in binary fields is somewhat more involved than modular square-root computations in prime fields, so our symbolic-manipulation algorithms are expected to be rather effective for binary fields.

References

1. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik, and S. Vanstone, 'Accelerated verification of ECDSA signatures', SAC 2005, LNCS Vol. 3897, 307–318, 2006.
2. ANSI, 'Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)', ANSI X9.62, approved January 7, 1999.
3. J. H. Cheon and J. H. Yi, 'Fast batch verification of multiple signatures', PKC 2007, LNCS Vol. 4450, 442–457, 2007.
4. L. Harn, 'Batch verifying multiple RSA digital signatures', Electronics Letters, Vol. 34, No. 12, 1219–1220, 1998.
5. M.-S. Hwang, I.-C. Lin, K.-F. Hwang, 'Cryptanalysis of the Batch Verifying Multiple RSA Digital Signatures', Informatica, Vol. 11, No. 1, 15–19, 2000.
6. D. Johnson and A. Menezes, 'The Elliptic Curve Digital Signature Algorithm (ECDSA)', International Journal on Information Security, Vol. 1, 36–63, 2001.
7. D. Naccache, D. M'Raihi, D. Rapheali and S. Vaudenay, 'Can D.S.A. be improved: Complexity trade-offs with the digital signature standard', EuroCrypt'94, LNCS Vol. 950, 77–85, 1994.
8. NIST, 'Digital Signature Standard (DSS)', http://csrc.nist.gov/publications/drafts/fips_186-3/Draft-FIPS-186-3%20_March2006.pdf, 2006.
9. NIST, 'Recommended elliptic curves for federal government use', http://csrc.nist.gov/groups/ST/toolkit/ documents/dss/NISTReCur.pdf, July 1999.
10. NIST, 'Secure Hash Standard (SHS)', http://csrc.nist.gov/publications/drafts/fips_180-3/draft_fips-180-3_June-08-2007.pdf, 2007.
11. PARI Group, 'PARI/GP Development Headquarters', http://pari.math.u-bordeaux.fr/, 2003–2008.
12. R. L. Rivest, A. Shamir and L. Adleman, 'A method for obtaining digital signatures and pubic-key cryptosystem', Communications of the ACM, Vol. 2, 120–126, 1978.
13. D. Shanks. 'Five number theoretic algorithms', Proceedings of the Second Manitoba Conference on Numerical Mathematics, 51–70, 1973.

Appendix

A Properties of R_x and R_y

Theorem 1. *R_x contains only even-degree monomials, and R_y contains only odd-degree monomials in the variables $y_1, y_2, \ldots, y_t$.*

Proof. We proceed by induction on the batch size $t \geqslant 1$. If $t = 1$ (case of individual verification), we have $R_x = r_1$ and $R_y = y_1$, for which the theorem evidently holds.

So assume that $t \geqslant 2$. We compute $R = \sum_{i=1}^{t} R_i$ as $R' + R''$ with $R' = \sum_{i=1}^{\tau} R_i$ and $R'' = \sum_{i=\tau+1}^{t} R_i$ for some τ in the range $1 \leqslant \tau \leqslant t-1$. Let $R' = (R'_x, R'_y)$ and $R'' = (R''_x, R''_y)$. The inductive assumption is that all non-zero terms of R'_x and R''_x are of even degrees (in $y_1, \ldots, y_\tau$ and $y_{\tau+1}, \ldots, y_t$, respectively), and all non-zero terms of R'_y and R''_y are of odd degrees.

We first symbolically compute $\lambda = (R''_y - R'_y)/(R''_x - R'_x)$ as a rational function. Clearing the variables y_i from the denominator multiplies both the numerator and the denominator of λ by polynomials of non-zero terms having even degrees. Every substitution of y_i^2 by the field element $r_i^3 + ar_i + b$ reduces the y_i-degree of certain terms by 2, so the parity of the degrees in these terms is not altered. Finally, λ becomes a polynomial with each non-zero term having odd degree. But then, $R_x = \lambda^2 - R'_x - R''_x$ is a polynomial with each non-zero term having even degree, whereas $R_y = \lambda(R'_x - R_x) - R'_y$ is a polynomial with each non-zero term having odd degree. Further substitutions of y_i^2 by $r_i^3 + ar_i + b$ to simplify R_x and R_y preserve these degree properties.

B Derivation of δ

For computing the number of roots $(r_1, r_2, \ldots, r_t)$ of $\det M = 0$, we treat $r_1, r_2, \ldots, r_t$ as symbols, and need to calculate an upper bound on the degree δ of each individual r_i. Without loss of generality, we compute an upper bound on the degree δ of r_1 in $\det M = 0$. To this effect, we first look at the expressions for R_x and R_y which are elements of $\mathbb{F}_q(r_1, r_2, \ldots, r_t)[y_1, y_2, \ldots, y_t]$. We can write $R_x = g_x/h$ and $R_y = g_y/h$, where g_x, g_y are polynomials in $\mathbb{F}_q[r_1, r_2, \ldots, r_t, y_1, y_2, \ldots, y_t]$, and the common denominator h is a polynomial in $\mathbb{F}_q[r_1, r_2, \ldots, r_t]$. Let η_t denote the maximum of the r_1-degrees in g_x, g_y and h. We first recursively derive an upper bound for η_t.

We compute $R = R' + R''$ with $R' = (R'_x, R'_y) = \sum_{i=1}^{\tau} R_i$ and $R'' = (R''_x, R''_y) = \sum_{i=\tau+1}^{t} R_i$, where $\tau = \lceil t/2 \rceil$. The r_1-degree of R' is η_τ, whereas the r_1-degree of R'' is 0. The initial r_1-degree of $\lambda = (R''_y - R'_y)/(R''_x - R'_x)$ is at most η_τ. Clearing y_1 from the denominator of λ changes the r_1-degree to $2\eta_\tau + 3$. Subsequent eliminations of $y_2, \ldots, y_t$ finally reduces λ with a y-free denominator. The maximum r_1-degree of this expression for λ is $2^{t-1}(2\eta_\tau + 3)$. Therefore, λ^2 has r_1-degree $\leqslant 2^t(2\eta_\tau + 3)$. Subsequent computations of $R_x = \lambda^2 - R'_x - R''_x$ and $R_y = \lambda(R'_x - R_x) - R'_y$ yield

$$\eta_t \leqslant (2^t + 2^{t-1})(2\eta_\tau + 3) + 2\eta_\tau \leqslant (2^t + 2^{t-1})(2\eta_\tau + 3) + 2\eta_\tau$$

with $\tau = \lceil t/2 \rceil$. Solving this recurrence gives the upper bound $\eta_t \leqslant 2^{2t+3\lceil \log_2 t \rceil + 2}$.

Now, we follow a sequence of squaring and monomial multiplication to convert $R_x = \alpha$ to a set of linear equations. If Δ_i is the r_1-degree of the i-th equation, we have

$$\Delta_1 = \eta_t,$$
$$\Delta_i \leqslant 2\Delta_{i-1} + 3 \text{ for } i \geqslant 2.$$

The recurrence relation pertains to the case of squaring. One easily checks that $\Delta_i \leqslant (\eta_t + 3)2^{i-1}$ for all $i \geqslant 1$. Finally, the r_1-degree of the equation $\det M = 0$ is

$$\delta \leqslant \Delta_1 + \Delta_2 + \cdots + \Delta_\mu \leqslant (\eta_t + 3)(2^\mu - 1) \leqslant \Big(2^{2t+3\lceil \log_2 t \rceil + 2} + 3\Big)\Big(2^{2^{t-1}-1} - 1\Big).$$

Notice that this is potentially a very loose upper bound for δ. In general, we avoid squaring. Multiplication by a monomial can increase the r_1-degree by 3 if the monomial contains y_1. If the monomial does not contain y_1, the r_1-degree does not increase at all. Nevertheless, this loose upper bound is good enough in the present context.

C Number of Roots of $\det M = 0$

Let us write the equation $\det M = 0$ as $D(r_1, r_2, \ldots, r_t) = 0$, where the r_i-degree of the multivariate polynomial D is $\leqslant \delta$ for each i. We assume that D is not identically zero. We plan to show that the maximum number $B^{(t)}$ of roots of D is $\leqslant t\delta q^{t-1}$. To that effect, we first write D as a polynomial in r_t:

$$D(r_1, r_2, \ldots, r_t) = D_\delta(r_1, r_2, \ldots, r_{t-1})r_t^\delta + D_{\delta-1}(r_1, r_2, \ldots, r_{t-1})r_t^{\delta-1} + \cdots + D_1(r_1, r_2, \ldots, r_{t-1})r_t + D_0(r_1, r_2, \ldots, r_{t-1}).$$

If D is not identically zero, at least one D_i is not identically zero. If $(r_1, r_2, \ldots, r_{t-1})$ is a common root of each D_i, appending any value of r_t gives a root of D. The maximum number of common roots of $D_0, D_1, \ldots, D_\delta$ is $B^{(t-1)}$. On the other hand, if $(r_1, r_2, \ldots, r_{t-1})$ is not a common root of all D_i, there are at most δ values of r_t satisfying $D(r_1, r_2, \ldots, r_t) = 0$. We, therefore, have

$$B^{(t)} \leqslant B^{(t-1)}q + (q^{t-1} - B^{(t-1)})\delta = (q - \delta)B^{(t-1)} + \delta q^{t-1}. \tag{14}$$

Moreover, we have

$$B^{(1)} \leqslant \delta. \tag{15}$$

By induction on t, one can show that $B^{(t)} \leqslant t\delta q^{t-1}$. This bound is rather tight, particularly for $\delta \ll q$ (as it happens in our cases of interest). A polynomial D satisfying equalities in (14) and (15) can be constructed as $D(r_1, r_2, \ldots, r_t) = \Delta(r_1)\Delta(r_2)\cdots\Delta(r_t)$, where Δ is a square-free univariate polynomial of degree δ, that splits over $\mathbb{F}_q$. By the principle of inclusion and exclusion (or by explicitly solving the recurrence (14)), we obtain the total number of roots of this D as

$$\delta t q^{t-1} - \binom{t}{2}\delta^2 q^{t-1} + \binom{t}{3}\delta^3 q^{t-3} - \cdots + (-1)^{t-1}\delta^t$$
$$= q^t - (q - \delta)^t \;=\; \delta(q^{t-1} + (q - \delta)q^{t-2} + (q - \delta)^2 q^{t-3} + \cdots + (q - \delta)^{t-1}).$$

If $\delta \ll q$, this count is very close to $t\delta q^{t-1}$. It remains questionable whether our equation $\det M = 0$ actually encounters this worst-case situation, but this does not matter, at least in a probabilistic sense.