Optimized Lattice Basis Reduction In Dimension 2, and Fast Schnorr and EdDSA Signature Verification

Thomas Pornin

NCC Group, `thomas.pornin@nccgroup.com`

18 April 2020

Abstract. We present an optimization of Lagrange's algorithm for lattice basis reduction in dimension 2. The optimized algorithm is proven to be correct and to always terminate with quadratic complexity; it uses more iterations on average than Lagrange's algorithm, but each iteration is much simpler to implement, and faster. The achieved speed is such that it makes application of the speed-up on ECDSA and EC Schnorr signatures described by Antipa *et al*[1] worthwhile, even for very fast curves such as Ed25519[3]. We applied this technique to signature verification in Curve9767[12], and reduced verification time by 30 to 33% on both small (ARM Cortex M0+ and M4) and large (Intel Coffee Lake with AVX2) architectures.

1 Introduction

Schnorr signatures[13] are an efficient signature algorithm that can be readily applied to most groups where discrete logarithm is a hard problem. Ed25519, defined in [3] and later extended to other curves and standardized under the generic name EdDSA[6], is an application of Schnorr signatures to an Edwards elliptic curve. Weierstraß curves can also be used, e.g. Curve9767[12]. Schnorr signature verification is typically more expensive that signature generation:

- Most of the cost of signature generation is computing point rB, where B is the conventional generator point for the prime order subgroup, and r is a per-signature secret scalar. Since B is fixed, precomputed tables can be used for many multiples of B, which greatly speeds up the operation.
- Conversely, signature verification relies on verifying the equation $h(sB - kA) = hR$, where point R and scalar s are part of the signature value, A is the public key, k is computed from the output of a hash function over the message, public key and R point, and h is the *cofactor*, i.e. the ratio between the curve order and the order of the subgroup generated by B (in practice, h is very small; for Ed25519, $h = 8$). Points R and A vary, precluding the use of precomputed tables.

A method for improving signature verification performance was described by Antipa *et al*[1]. The main idea is that the verification equation is equivalent to checking that:

$$h((\delta s)B - (\delta k)A - \delta R) = \mathbb{O}$$

for any value δ invertible modulo the subgroup order n (which is prime). The products δs and δk can be computed modulo n. By splitting $\delta s \bmod n$ into two halves e_0 and e_1 (say, of $m/2$

bits each, with the subgroup order n being an m-bit integer), that equation can be rewritten as:

$$h(e_0B + e_1 2^{m/2}B - (\delta k \bmod n)A - \delta R) = \mathbb{O}$$

When computing a linear combination of curve points, the well-known "Shamir's trick" (in fact due to Straus[14]) allows mutualizing all the point doublings in a double-and-add algorithm. In the case of $sB - kA$, both s and k have about the same size as n (i.e. about m bits, in our notations), and the double-and-add algorithm will involve about m doublings, and some extra additions. The number of extra additions can be reduced through a number of techniques (windows, sliding windows, NAF recoding...) and the m doublings dominate the algorithm cost.

In our rewritten equation, we have a linear combination of four points, two of which are fixed and thus known in advance (B and $2^{m/2}B$), and the other two being obtained dynamically (A and R). Coefficients e_0 and e_1 have size $m/2$ bits (half that of the prime subgroup order n). Efficiency comes from finding a value δ which is such that both δ and $\delta k \bmod n$ are "small", i.e. about $m/2 + 1$ bits each. If such a δ is found, then there will be only about $m/2$ point doublings instead of m, i.e. halving the cost of the most expensive part of the computation (the number of extra additions remains about the same as in the initial verification equation, since there are now four points instead of two, but with twice as small coefficients).

A method for finding an appropriate δ is found in [4]; in a nutshell, it uses an extended Euclidean GCD algorithm, and stops when the intermediate value has shrunk to about half the size of n. This *heuristically* finds a good δ value.

The Ed25519 article ([3], page 14) explicitly considers the Antipa *et al* method, and rejects it, on the grounds that the Euclidean GCD algorithm implies a number of divisions, which are cumbersome and expensive to implement, making the cost of the computation of δ higher than the savings obtained through reduction of the number of point doublings.

In this note, we describe an alternate algorithm for computing δ, which is efficient enough to make the optimization worthwhile, even for fast signature algorithms such as Curve25519. Our algorithm can be viewed as a binary version of Lagrange's algorithm for reduction of a lattice basis in dimension two; in particular, no division is used, and only two plain integer multiplications are involved, the inner loop consisting only of efficient linear operations (additions, subtractions, and left shifts).

Lagrange's algorithm, and our new algorithm, are described in section 2. Some implementation notes and speed benchmarks are given in section 3.

2 Lattice Basis Reduction

2.1 Notations

We use a bold font to designate vectors, e.g. $\mathbf{u}$ and $\mathbf{v}$. All vectors are in the vector space $\mathbb{R}^e$ for some integer e (in most of the article, we will focus on $e = 2$); moreover, apart from the initial presentation of lattices, all vector coordinates will be integers.

For an integer x, we denote $\text{len}(x)$ the minimum size of the binary representation of x, using two's complement for negative values, and excluding the sign bit. If $x \geq 0$, then $\text{len}(x)$ is the smallest integer $m \geq 0$ such that $x < 2^m$. If $x < 0$, then $\text{len}(x) = \text{len}(-x - 1)$.

There are other possible definitions for len, depending on whether $\text{len}(2^m)$ is defined to be m or $m + 1$, and similarly for $\text{len}(-2^m)$. Our definition implies that $\text{len}(2^m) = m + 1$ but $\text{len}(-2^m) = m$, because it maps well to an implementation that uses two's complement for negative values. All four possible combinations actually work in our algorithm.

2.2 Lattices

We define here lattices and recall some known results. For ampler explanations and proofs, refer to [11].

Let $e \geq 2$ an integer; we consider the vector space $\mathbb{R}^e$. A *lattice* is a discrete subgroup of $\mathbb{R}^e$. It can be shown that any lattice L has a basis consisting of e' linearly independent vectors $(\mathbf{b}_i)$ (hence $e' \leq e$), such that:

$$L = \left\{ \sum_{i=1}^{e} x_i \mathbf{b}_i \,\middle|\, (x_1, x_2, \dots x_{e'}) \in \mathbb{Z}^{e'} \right\}$$

A lattice has, in general, infinitely many bases; however, they all contain the same number e' of vectors, which is the *rank* of the lattice. In this paper, we will consider only lattices of maximal rank (i.e. $e = e'$), and in particular the case of lattices of rank 2 in a space of dimension $e = 2$ (hereafter called "lattices of dimension 2").

If B is the $e \times e$ matrix whose column vectors are a basis of L, then the *volume* of the lattice L is the square root of the determinant of BB^t (where B^t is the transpose of B). It can be shown that this quantity does not depend on which basis is used for a given lattice. It is denoted $\text{vol}(L)$.

It shall be noted that if the elements of a lattice basis only have integer coordinates, then all the lattice elements, and therefore all elements of any other basis of the lattice, will have only integer coordinates.

For two vectors $\mathbf{u}$ and $\mathbf{v}$, let $\langle \mathbf{u}, \mathbf{v} \rangle$ be their inner product; we denote $N(\mathbf{u}) = \langle \mathbf{u}, \mathbf{u} \rangle$. $N(\mathbf{u})$ is the square of the Euclidean norm of $\mathbf{u}$. A well-known problem is to find a *shortest non-zero vector* in a lattice. The minimal length of non-zero vectors in a lattice is well-defined, and there is only a finite number of vectors in the lattice whose length is exactly the minimal length[1]. When the dimension is large, finding a shortest vector is a hard problem, but in very low dimensions, efficient algorithms are known[11]. Hermite proved[7] in 1850 that there exists a constant γ_e that depends only on the dimension e, such that if $\mathbf{s}$ is a shortest non-zero vector of a lattice L, $N(\mathbf{s}) \leq \gamma_e \text{vol}(L)^{2/e}$. Hermite proved that $\gamma_e \leq (4/3)^{(e-1)/2}$, and Minkowski later showed[10] that $\gamma_e \leq e$. The exact value of γ_e is known only for small dimensions; in particular, $\gamma_2 = 2/\sqrt{3} \approx 1.16$.

Application To Schnorr and EdDSA Signature Verification. As explained in the introduction, the problem we are trying to solve is, given a prime n and an integer $0 \leq k < n$, finding two integers d_0 and d_1 such that $|d_0|$ and $|d_1|$ are small, and $d_0 = d_1 k \bmod n$ (note that either or both d_0 and d_1 can be negative). We can rewrite that problem as an instance of finding a short vector in a lattice of dimension two (in the introduction, we used the letter δ to

[1]If $\mathbf{u}$ is a shortest non-zero vector, then so is $-\mathbf{u}$; therefore, the solution is never unique.

designate d_1; we switch the notation to emphasize the vector interpretation of the problem). Indeed, we define the two vectors:

$$\mathbf{u} = (n, 0)$$
$$\mathbf{v} = (k, 1)$$

These two vectors are linearly independent, and therefore are the basis for a lattice L of dimension two:

$$L = \left\{\alpha\mathbf{u} + \beta\mathbf{v} \mid (\alpha, \beta) \in \mathbb{Z}^2\right\}$$

The volume of this lattice is $\mathrm{vol}(L) = \sqrt{n^2} = n$.

Any vector $\mathbf{c} = (c_0, c_1)$ in the lattice L is such that $c_0 = kc_1 + \alpha n$ for some integer α, and, conversely, any pair of integers (c_0, c_1) such that $c_0 = kc_1 \bmod n$ is part of the lattice (it suffices to set $\beta = c_1$ and $\alpha = (c_0 - kc_1)/n$). The vector $\mathbf{d} = (d_0, d_1)$ we are looking for is thus an element of the lattice, and we want it to be as short as possible. Hermite's theorem shows that a shortest non-zero vector in that lattice will be such that:

$$d_0^2 + d_1^2 = N(\mathbf{d}) \leq \gamma_2 \mathrm{vol}(L) \approx 1.16n$$

It follows that there must exist a non-zero vector $\mathbf{d}$ in the lattice L such that $|d_0|$ and $|d_1|$ are both at most $\sqrt{1.16n}$, i.e. about half the size of n. In particular, if $n < 2^m$ (an m-bit integer), then we can always find integers d_0 and d_1 of at most $\lfloor(m+2)/2\rfloor$ bits each (in absolute value).

2.3 Lagrange's Algorithm

In 1773, Lagrange described an efficient algorithm to find a shortest vector of a lattice in dimension two[8] (that algorithm was independently rediscovered and published by Gauss in 1801[5], and is sometimes called Gauss's algorithm). The algorithm can be considered to be an extension of the classic Euclidean GCD algorithm. It is described in algorithm 1.

Algorithm 1 Lagrange's algorithm

```
Require: $(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^2 \times \mathbb{Z}^2$ a basis for a lattice $L$ of dimension two
Ensure: A size-reduced basis of $L$
1: if $N(\mathbf{u}) < N(\mathbf{v})$ then
2:     swap $\mathbf{u}$ with $\mathbf{v}$
3: repeat
4:     $q \leftarrow \lfloor\langle\mathbf{u}, \mathbf{v}\rangle / N(\mathbf{v})\rceil$      ▷ Division rounded to a nearest integer.
5:     $\mathbf{u} \leftarrow \mathbf{u} - q\mathbf{v}$
6:     swap $\mathbf{u}$ with $\mathbf{v}$
7: until $N(\mathbf{u}) \leq N(\mathbf{v})$
8: return $(\mathbf{v}, \mathbf{u})$      ▷ $\mathbf{v}$ is a shortest non-zero vector in $L$.
```

Intuitively, Lagrange's algorithm works as follows: given a lattice basis $(\mathbf{u}, \mathbf{v})$, the pair $(\mathbf{v}, \mathbf{u} - q\mathbf{v})$ is another basis, for any integer q (we recognize here a step similar to the Euclidean GCD algorithm). We converge on a short vector by applying that transformation repeatedly, until it no longer works, i.e. the basis cannot be made any shorter in this way. If we were to allow a non-integral q, then the shortest vector $\mathbf{u} - q\mathbf{v}$ would correspond to the orthogonal projection of $\mathbf{u}$ along the direction of $\mathbf{v}$, with value $q = \langle \mathbf{u}, \mathbf{v} \rangle / N(\mathbf{v})$, and no further size reduction would be possible once the basis has been made orthogonal. Since q must be an integer (otherwise, the modified basis is no longer a basis for the source lattice L), the algorithm rounds the fraction to a nearest integer, yielding a new basis which is not fully orthogonal, but "more orthogonal" than the previous one.

Beyond this intuitive description, it is not obvious why this algorithm converges on a shortest vector in the lattice, and why the overall complexity is quadratic. Extensive analysis of this algorithm can be found in [11]. Lagrange's algorithm has the following properties:

- When the algorithm terminates, $\mathbf{v}$ is a shortest non-zero vector of the lattice, and $\mathbf{u}$ is a shortest vector of the lattice among those which are linearly independent with $\mathbf{v}$. The basis $(\mathbf{u}, \mathbf{v})$ is then said to be *size-reduced*.
- If, at the beginning of an iteration, $|\langle \mathbf{u}, \mathbf{v} \rangle| \leq N(\mathbf{v})/2$, then $(\mathbf{u}, \mathbf{v})$ is size-reduced and the algorithm terminates. Conversely, the algorithm does *not* terminate until the basis is size-reduced.
- It may happen that $|q| = 1$ only at the first and last iterations of the algorithm (this is due to the greedy nature of the algorithm, see section 3.2 of [11]).

Algorithm 1 seems to require the computation of $\langle \mathbf{u}, \mathbf{v} \rangle$ and $N(\mathbf{v})$ at each iteration, implying multiplications of possibly large integers. However, the successive values of $\langle \mathbf{u}, \mathbf{v} \rangle$, $N(\mathbf{u})$ and $N(\mathbf{v})$ can be computed more efficiently by noting that:

$$\begin{aligned} N(\mathbf{u} - q\mathbf{v}) &= N(\mathbf{u}) + q^2 N(\mathbf{v}) - 2q\langle \mathbf{u}, \mathbf{v} \rangle \\ \langle \mathbf{v}, \mathbf{u} - q\mathbf{v} \rangle &= \langle \mathbf{u}, \mathbf{v} \rangle - qN(\mathbf{v}) \end{aligned}$$

In algorithm 2, we describe Lagrange's algorithm again, but with explicit updates that follow these rules. The value $N(\mathbf{u})$, $N(\mathbf{v})$ and $\langle \mathbf{u}, \mathbf{v} \rangle$ are kept in variables N_u, N_v and p, respectively, which are adjusted with the above formulas whenever $\mathbf{u}$ or $\mathbf{v}$ is changed; notably, when $\mathbf{u}$ and $\mathbf{v}$ are swapped with each other, $N(\mathbf{u})$ and $N(\mathbf{v})$ are swapped accordingly (since $\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle$, the variable p does not need to be modified when swapping $\mathbf{u}$ and $\mathbf{v}$).

This description shows that all operations in an iteration can be expressed as additions, subtractions, multiplications by q, and Euclidean division (rounded). The values q are usually small, making these operations efficient on average. The overall complexity of the algorithm is $O((\log \|\mathbf{u}\|)^2)$, i.e. quadratic in the norm of the largest vector of the input basis; intuitively, each iteration reduces the norm of the larger vector ($\mathbf{u}$) by $O(\log q)$ bits, so that larger q induce better reduction, and the cumulative cost of all the multiplications is quadratic.

In practice, this algorithm is cumbersome to implement, because the value q, while normally small, *can* be large, requiring the implementation of multiplication and division over large integers. Moreover, even when q is small, multiplication of a big integer by a small q is slower than addition, by a constant factor that depends on the underlying hardware abilities (cost of multiplication relatively to addition) but is not negligible. These characteristics make Lagrange's algorithm less interesting for application of the method of Antipa *et al* to signature verification, especially for fast curves.

Algorithm 2 Lagrange's algorithm (with efficient updates of $\langle \mathbf{u}, \mathbf{v} \rangle$, $N(\mathbf{u})$ and $N(\mathbf{v})$)

```
Require: $(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^2 \times \mathbb{Z}^2$ a basis for a lattice $L$ of dimension two
Ensure: A size-reduced basis of $L$
 1: $N_u \leftarrow N(\mathbf{u})$
 2: $N_v \leftarrow N(\mathbf{v})$
 3: $p \leftarrow \langle \mathbf{u}, \mathbf{v} \rangle$
 4: if $N_u < N_v$ then
 5:     swap u with v
 6:     swap $N_u$ with $N_v$
 7: repeat
 8:     $q \leftarrow \lfloor p/N_v \rceil$                      ▷ Division rounded to a nearest integer.
 9:     $\mathbf{u} \leftarrow \mathbf{u} - q\mathbf{v}$
10:     $N_u \leftarrow N_u + q^2 N_v - 2qp$   ▷ The value of $N_u$ is adjusted to $N(\mathbf{u})$ for the new value of u.
11:     $p \leftarrow p - qN_v$                 ▷ The value of $p$ is adjusted to $\langle \mathbf{u}, \mathbf{v} \rangle$ for the new value of u.
12:     swap u with v
13:     swap $N_u$ with $N_v$          ▷ Swapping $u$ and $v$ does not change the value of $\langle \mathbf{u}, \mathbf{v} \rangle$.
14: until $N_u \leq N_v$
15: return (v, u)                              ▷ v is a shortest non-zero vector in $L$.
```

2.4 Improved Algorithm

We define in algorithm 3 a variant of Lagrange's algorithm which is much easier to implement, and in practice more efficient, since it uses only additions, subtractions and left shifts in the main loop. The main idea is to force q to be a power of two ($q = 2^s$ for some integer $s \geq 0$).

Like in algorithm 2, values of $N(\mathbf{u})$, $N(\mathbf{v})$ and $\langle \mathbf{u}, \mathbf{v} \rangle$ are cached in variables N_u, N_v and p, and updated with explicit operations.

The two main differences with Lagrange's algorithm are the following:

- The quotient q is forced to $\pm 2^s$ for some integer s.
- An iteration does not necessarily reduce $\mathbf{u}$ to a size lower than that of $\mathbf{v}$. Therefore, the swap is made conditional to the actual norms. Consequently, the exit test (on line 8) cannot use a failure to reduce $\mathbf{u}$ to a size lower than $\mathbf{v}$; instead, the algorithm exits when a size-reduced basis is obtained.

As we will see in section 2.5, in the case of optimizing EdDSA or Schnorr signature verification, where we only need a *short* vector, not necessarily a *shortest* vector, the test on line 8 can be replaced with a much cheaper test on the bit length of N_v, bit length which is computed on line 10.

Correctness. Like Lagrange's algorithm, the improved algorithm modifies $(\mathbf{u}, \mathbf{v})$ only by swapping $\mathbf{u}$ with $\mathbf{v}$, and by replacing $\mathbf{u}$ with $\mathbf{u} - q\mathbf{v}$ for some integer q; both operations naturally lead to another basis for the same lattice L. Moreover, the exit condition is such that the algorithm can only return a size-reduced basis, which implies that its smaller vector ($\mathbf{v}$) is necessarily a shortest non-vector in the lattice. What remains to be proven is that the algorithm always terminates.

Algorithm 3 Improved algorithm

```
Require: $(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^2 \times \mathbb{Z}^2$ a basis for a lattice $L$ of dimension two
Ensure: A short non-zero vector of $L$
 1: $N_u \leftarrow N(\mathbf{u})$
 2: $N_v \leftarrow N(\mathbf{v})$
 3: $p \leftarrow \langle \mathbf{u}, \mathbf{v} \rangle$
 4: loop
 5:     if $N_u < N_v$ then
 6:         swap $\mathbf{u}$ with $\mathbf{v}$
 7:         swap $N_u$ with $N_v$
 8:     if $2|p| \le N_v$ then                ▷ This test matches when the basis is size-reduced.
 9:         return $(\mathbf{v}, \mathbf{u})$
10:     $s \leftarrow \max(0, \mathrm{len}(p) - \mathrm{len}(N_v))$
11:     if $p > 0$ then
12:         $\mathbf{u} \leftarrow \mathbf{u} - 2^s\mathbf{v}$
13:         $N_u \leftarrow N_u + 2^{2s}N_v - 2^{s+1}p$
14:         $p \leftarrow p - 2^s N_v$
15:     else
16:         $\mathbf{u} \leftarrow \mathbf{u} + 2^s\mathbf{v}$
17:         $N_u \leftarrow N_u + 2^{2s}N_v + 2^{s+1}p$
18:         $p \leftarrow p + 2^s N_v$
```

Consider the situation when reaching line 8. Define:

$$a = \mathrm{len}(\langle \mathbf{u}, \mathbf{v} \rangle)$$
$$b = \mathrm{len}(N(\mathbf{v}))$$

Since $N(\mathbf{v}) > 0$, we have:

$$2^{b-1} \le N(\mathbf{v}) \le 2^b - 1$$

Note that if $\langle \mathbf{u}, \mathbf{v} \rangle = 0$, then the exit test matches and the algorithm exits. Therefore, line 10 may be reached only if $\langle \mathbf{u}, \mathbf{v} \rangle \neq 0$. There are now three possible cases to consider:

Case 1: $a \ge b$ **and** $\langle \mathbf{u}, \mathbf{v} \rangle > 0$

We have:

$$2^{a-1} \le \langle \mathbf{u}, \mathbf{v} \rangle \le 2^a - 1$$

which implies that:

$$-2^{a-1} + 2^{a-b} \le \langle \mathbf{u}, \mathbf{v} \rangle - 2^s N(\mathbf{v}) \le 2^{a-1} - 1$$

which means that the bit length of $|\langle \mathbf{u}, \mathbf{v} \rangle|$ at the end of the iteration must be at most $a - 1$, i.e. at least one bit shorter than its previous value.

Case 2: $a \ge b$ **and** $\langle \mathbf{u}, \mathbf{v} \rangle < 0$

A similar analysis applies:

$$-2^a \le \langle \mathbf{u}, \mathbf{v} \rangle \le -2^{a-1} - 1$$

and therefore:

$$-2^{a-1} \leq \langle \mathbf{u}, \mathbf{v} \rangle - 2^s N(\mathbf{v}) \leq 2^{a-1} - 2^{a-b} - 1$$

which again implies that $|\langle \mathbf{u}, \mathbf{v} \rangle|$ has been shortened by at least one bit.

Case 3: $a < b$

We have $s = 0$, and $q = 1$ if $\langle \mathbf{u}, \mathbf{v} \rangle > 0$, -1 otherwise. We know that $|\langle \mathbf{u}, \mathbf{v} \rangle| > N(\mathbf{v})/2$, since the exit test on line 8 was not matched. This implies that:

$$|\langle \mathbf{u}, \mathbf{v} \rangle - qN(\mathbf{v})| < N(\mathbf{v})/2 < |\langle \mathbf{u}, \mathbf{v} \rangle|$$

Therefore, $|\langle \mathbf{u}, \mathbf{v} \rangle|$ is made strictly lower during this iteration, and the algorithm makes progress; we cannot get forever in this case.

We can furthermore show that if we reach this case, then the next iteration will either be the last one, or it will match one of the two other cases where $|\langle \mathbf{u}, \mathbf{v} \rangle|$ is made shorter by at least one bit. Consider that if Lagrange's algorithm were invoked on the current basis $(\mathbf{u}, \mathbf{v})$, then it would use a value q such that $|q| = 1$, and the sign of q would be identical to that of $\langle \mathbf{u}, \mathbf{v} \rangle$. In that situation, algorithm 3 uses the exact same value of $q = \pm 2^s = \pm 1$, and, as such computes the same new basis as Lagrange's algorithm. However, if Lagrange's algorithm uses $|q| = 1$ for two consecutive steps, then the second step is the last one and the resulting basis is size-reduced. Correspondingly, if algorithm 3 reaches the case of $a < b$ in any iteration, then either the next iteration will be in the same case, and that will yield a size-reduced basis and terminate the algorithm; or the next iteration will use one of the two other cases where $a \geq b$.

From this analysis, it follows that $|\langle \mathbf{u}, \mathbf{v} \rangle|$ must be reduced by at least one bit for every two iterations, unless the algorithm reaches the exit condition after the second iteration. None of the iterations ever allows $|\langle \mathbf{u}, \mathbf{v} \rangle|$ to grow. Therefore, the total number of iterations cannot be greater than $1 + 2\lceil \log |\langle \mathbf{u}, \mathbf{v} \rangle| \rceil$ (for the vectors $\mathbf{u}$ and $\mathbf{v}$ originally used as input, and with log denoting the base-2 logarithm). This proves that the algorithm always terminates, and, by construction, returns a size-reduced basis whose smaller vector is a shortest non-zero vector of the lattice L.

Complexity. The analysis above implies that the number of iterations is necessarily at most $1 + 2\lceil \log(|\langle \mathbf{u}, \mathbf{v} \rangle|) \rceil$. If $\mathbf{u}$ is the largest vector of the input $(\mathbf{u}, \mathbf{v})$ basis, then $|\langle \mathbf{u}, \mathbf{v} \rangle| \leq N(\mathbf{u})$; therefore, the number of iterations is at most $1 + 4\lceil \log \|\mathbf{u}\| \rceil$, i.e. linear in the size of the largest input vector.

Suppose that, at a given iteration, $\langle \mathbf{u}, \mathbf{v} \rangle > 0$, and $a \geq b$ (where $a = \text{len}(\langle \mathbf{u}, \mathbf{v} \rangle)$ and $b = \text{len}(N(\mathbf{v}))$, as in the previous analysis). Then:

$$\begin{aligned} 2^{2s} N(\mathbf{v}) - 2^{s+1} \langle \mathbf{u}, \mathbf{v} \rangle &\leq 2^{2(a-b)} 2^b - 2^{a-b+1} 2^{a-1} \\ &\leq 0 \end{aligned}$$

i.e. $N(\mathbf{u})$ (in variable N_u in the algorithm) is updated by adding a value which is negative or zero. Therefore, the size of $\mathbf{u}$, the larger vector in the current $(\mathbf{u}, \mathbf{v})$ basis, cannot be increased. A similar analysis holds when $\langle \mathbf{u}, \mathbf{v} \rangle < 0$. And when $a < b$, as in the previous analysis, our algorithm follows the same steps as Lagrange's algorithm, in which the size of the larger vector in the basis cannot be increased either. We can thus conclude that, at all steps of the algorithm:

- $|u_0|$, $|u_1|$, $|v_0|$ and $|v_1|$ can never be greater than the initial norm $\|\mathbf{u}\|$ (where $\mathbf{u} = (u_0, u_1)$ and $\mathbf{v} = (v_0, v_1)$).
- $N(\mathbf{u})$, $N(\mathbf{v})$ and $|\langle \mathbf{u}, \mathbf{v}\rangle|$ can never be greater than the initial squared norm $N(\mathbf{u})$.

Therefore, all operations in each iteration, which are linear in the size of the operands, have cost at most $O(\log \|\mathbf{u}\|)$. Since there are $O(\log \|\mathbf{u}\|)$ iterations (at most), the overall complexity of our algorithm is $O((\log \|\mathbf{u}\|)^2)$, i.e. quadratic.

It shall be noted that, in practice:

- All values (coordinates of $\mathbf{u}$ and $\mathbf{v}$, squared norms $N(\mathbf{u})$ and $N(\mathbf{v})$, inner product $\langle \mathbf{u}, \mathbf{v}\rangle$) decrease in size as the algorithm progresses. The size reduction can be leveraged by implementations to reduce cost.
- The *average* number of iterations is lower than $4\lceil \log \|\mathbf{u}\|\rceil$. We measured over 500 million instances on random inputs in an EdDSA context ($\mathbf{u} = (n, 0)$ and $\mathbf{v} = (k, 1)$, with $n \approx 2^{252}$ and k chosen uniformly in $0 \dots n-1$); the average number of iterations was 97.867 with a standard deviation of 6.222. The observed minimum and maximum numbers of iterations were 64 and 135, respectively. It is probable that the upper bound computed above ($4 \times 252 = 1004$) is in fact unreachable in practice.

2.5 Application To Schnorr and EdDSA

When applying algorithm 3 to Schnorr and EdDSA signature verification, a number of extra optimizations are possible. We recall that, in that case, the input $(\mathbf{u}, \mathbf{v})$ to the algorithm is:

$$\mathbf{u} = (n, 0)$$
$$\mathbf{v} = (k, 1)$$

for the curve subgroup order n, and some integer k such that $0 \le k < n$. The volume of the lattice is $\mathrm{vol}(L) = n$ and a shortest non-zero vector has coordinates not exceeding (in absolute value) $\sqrt{\gamma_2 n} \approx \sqrt{1.16n}$.

Applicable optimizations include the following:

- Since three coordinates of the basis are statically known (only k may change between two invocations of the algorithm), some values can be precomputed, e.g. the initial value of $N(\mathbf{u}) = n^2$.
- We do not need a size-reduced basis; we only need one vector. Moreover, there is no crucial need for obtaining a *shortest* vector, only a *short enough* vector. We can thus use a looser exit condition based on the current bit length of $N(\mathbf{v})$. In particular, that bit length must already be obtained for the computation of s; reusing it makes the exit test extremely cheap.
- We already know the maximum length r, in bits (including the sign bit), of the coordinates of the returned short vector: $r = \lfloor(\mathrm{len}(n)+4)/2\rfloor$; in practice $r = 128$ for a 252-bit or 253-bit modulus n. Since all operations on $\mathbf{u}$ and $\mathbf{v}$ are additions, subtractions and left shifts, i.e. only operations with low-to-high propagation, we can keep the coordinates of $\mathbf{u}$ and $\mathbf{v}$ over r bits only, discarding upper bits. We will denote with "$(\mathrm{mod}^{\pm} 2^r)$" such computations[2].

[2] A practical implementation can use a larger value r if that is more convenient in a given situation; truncation is allowed but not mandatory.

Algorithm 4 shows these optimizations.

Algorithm 4 Improved algorithm applied to Schnorr and EdDSA signature verification

```
Require: Prime integer $n$, integer $k$ such that $0 \leq k < n$        ▷ $n$ is known statically.
Ensure: A short non-zero vector $(v_0, v_1)$ such that $v_0 = v_1 k \bmod n$
 1: $N_u \leftarrow n^2, N_v \leftarrow k^2 + 1, p \leftarrow nk$        ▷ $N_u$ is known statically.
 2: $r \leftarrow \lfloor(\mathrm{len}(n) + 4)/2\rfloor, t \leftarrow \mathrm{len}(n) + 1$        ▷ $t$ is the target length for $N(\mathbf{v})$.
 3: $(u_0, u_1) \leftarrow (n, 0) \pmod{^{\pm} 2^r}$
 4: $(v_0, v_1) \leftarrow (k, 1) \pmod{^{\pm} 2^r}$        ▷ Coordinates are represented over $r$ bits.
 5: loop
 6:     if $N_u < N_v$ then
 7:         swap $(u_0, u_1)$ with $(v_0, v_1)$
 8:         swap $N_u$ with $N_v$
 9:     if $\mathrm{len}(N_v) \leq t$ then        ▷ This test matches when $(v_0, v_1)$ is short enough.
10:         return $(v_0, v_1)$
11:     $s \leftarrow \max(0, \mathrm{len}(p) - \mathrm{len}(N_v))$        ▷ $\mathrm{len}(N_v)$ was already computed on line 9.
12:     if $p > 0$ then
13:         $(u_0, u_1) \leftarrow (u_0 - 2^s v_0, u_1 - 2^s v_1) \pmod{^{\pm} 2^r}$
14:         $N_u \leftarrow N_u + 2^{2s} N_v - 2^{s+1} p$
15:         $p \leftarrow p - 2^s N_v$
16:     else
17:         $(u_0, u_1) \leftarrow (u_0 + 2^s v_0, u_1 + 2^s v_1) \pmod{^{\pm} 2^r}$
18:         $N_u \leftarrow N_u + 2^{2s} N_v + 2^{s+1} p$
19:         $p \leftarrow p + 2^s N_v$
```

As was analyzed, N_u and N_v decrease regularly through the algorithm, and thus do not need more bits than the initial value $N_u = n^2$, which is statically known. N_u and N_v are always nonnegative, and therefore do not need an extra sign bit. The inner product p, on the other hand, can become negative. Since $|p| \leq N_u$, only one more bit than for the representation of N_u is sufficient.

Some extra analysis can help with reducing size. In particular, if working with a 256-bit curve such as P-256 or secp256k1, with a prime order n very close to 2^{256}, the analysis above would mean that 513 bits are needed, in all generality, to encode p, which is inconvenient if, for instance, using 64-bit limbs: this would require an extra ninth limb. However, the following can easily be seen:

- The initial value of p is known to be nonnegative. Therefore, the first iteration does not formally need to store the sign bit of p.
- After the first iteration, the value of p is necessarily at least one bit shorter (in absolute value) than the initial maximum n^2, leaving room for a sign bit.

Thus, provided that the first iteration uses a special treatment for the sign of p, an m-bit curve can be handled with a storage size of $2m$ bits for each of N_u, N_v and p.

3 Implementation Notes and Benchmarks

We implemented our improved algorithm (algorithm 4) for faster Schnorr signature verification over Curve9767[12], a prime-order Weierstraß curve well suited to embedded systems. Curve9767 source code is available on:

`https://github.com/pornin/curve9767`

Several implementations are provided, for different architectures:

– `src/scalar_ref.c`: reference implementation in portable C code.
– `src/scalar_cm0.s`: optimized assembly for ARM Cortex M0 and M0+ (ARMv6-M architecture).
– `src/scalar_cm4.s`: optimized assembly for ARM Cortex M4 (ARMv7-M architecture).
– `src/scalar_amd64.c`: optimized assembly for x86 in 64-bit mode (the 128-bit type `__int128` and carry-propagation intrinsics are used).

Curve9767 order is $n \approx 2^{251.82}$, i.e. a 252-bit integer. The two short vector coordinates will thus fit over $r = 128$ bits each, including the sign bit. Values N_u, N_v and p use up to 505 bits each, including the sign bit (as pointed out in section 2.5, N_u and N_v are always positive, and the extra bit for p is not really needed; thus, 504 bits would be sufficient).

Integer Representation. Integer values are represented as sequences of *limbs*; each limb has size w bits, and is a "digit" in base 2^w. Additions and subtractions are performed on a limb-by-limb basis. In the reference portable C code, we use $w = 30$, so that intermediate values fit in a 32-bit type (`uint32_t`); thus, N_u, N_v and p use 17 limbs each, and the vector coordinates u_0, u_1, v_0 and v_1 use 5 limbs each. In the specialized implementations, we can leverage the carry/borrow flag offered by the CPU (but inaccessible from portable C code); thus, we can use $w = 32$ for the ARM implementations (16 limbs for N_u, N_v and p, 4 limbs for u_0, u_1, v_0 and v_1), and $w = 64$ for the 64-bit x86 implementation (8 limbs for N_u, N_v and p, 2 limbs for u_0, u_1, v_0 and v_1).

Since N_u, N_v and p decrease over the course of the algorithm, we can switch to code with fewer limbs when these values have become short enough. At each iteration, $\mathbf{u}$ and $\mathbf{v}$ are conditionally swapped to ensure that $\mathbf{u}$ is the larger vector; we can thus use the size of N_u to decide when the alternate code with fewer limbs is safe. This strategy is applied in the x86 code, with a switch to 6 limbs then $N_u < 2^{383}$ (we know that $|p| \leq N_u$, but we need an extra sign bit, hence a threshold at 383 instead of 384 bits). In the ARM implementations, the number of limbs is similarly decreased when possible, but always kept even for implementation reasons (the addition, subtraction and shift loops use iterations that process two limbs at a time, to better optimize memory accesses).

Bit Length. Our algorithm must evaluate the exact bit length of big integers (N_v and p). This requires locating the most significant non-zero limb value, then the most significant non-zero bit within that limb value. In all generality, finding the position of the highest non-zero bit in a word of w bits can be done with a dichotomic search, with $\log w$ steps. This is how we implement it in the reference C code, and in the ARM Cortex M0 implementation. Some architectures offer dedicated opcodes that can make that operation much faster: on the ARM Cortex M4, we use the `clz` opcode, which returns the number of leading zeros in a 32-bit word in a single clock cycle; on x86, the `lzcnt` opcode (`_lzcnt_u64()` intrinsic function) offers the same service.

Variable Time Processing. Cryptographic implementations that run on physical systems, as opposed to abstract mathematical models of computing machines, may leak information about the values they process through side channels. In particular, side channels based on timing measures can often be efficiently exploited, possibly from remote view points. Implementations that do not have such side channel information leaks are said to be *constant-time*. In particular, constant-time code should not contain conditional jumps that depend on secret data, or make memory accesses at addresses that depend on secret data. Our lattice basis reduction algorithm is inherently *not* constant-time: the conditional swap of $\mathbf{u}$ and $\mathbf{v}$, the sign of q, the switch to shorter representations... all use conditional jumps that depend on the reduced value. This is usually considered tolerable for signature verification, because only the public key is used, and it is, by definition, non-secret.

One can imagine contrived scenarios in which a constant-time signature verification algorithm must remain secret: for instance, the signed message is secret but of low entropy (e.g. a human-memorable password), and the second half of the signature (the s value in a signature (R, s)) is somehow conveyed secretely to the verifier along with the signed message. In that case, the attacker cannot obtain enough information to run an offline dictionary attack on the message, but timing-based side channels in the signature verification algorithm can yield such information. Such scenarios are not very plausible, but we nonetheless provide *two* Schnorr signature verification implementations in the Curve9767 code:

- `curve9767_sign_verify()` is constant-time and does not use lattice basis reduction.
- `curve9767_sign_verify_vartime()` is not constant-time; it uses our improved lattice basis reduction algorithm and is faster than the constant-time implementation.

Point Multiplication Algorithms. The constant-time signature verification algorithm uses Straus's algorithm. The function computes $sB - kA$ with s and k having the same size as the curve order n; two *windows* are computed with multiples of A and B (multiples of B are precomputed, since B is fixed, but multiple of A must be computed at the start of the function and stored in a temporary stack buffer). For instance, the ARM implementations use 4-bit windows; every four point doublings, a point jA and a point $j'B$ are added, with $-8 \leq j < 8$ and $-8 \leq j' < 8$. Since point negation is inexpensive to compute dynamically, only 8 points jA ($1 \leq j \leq 8$) need to be stored in the stack.

When applying the improved lattice basis reduction algorithm, we now have to compute a linear combination of *four* points, as explained in section 1. Two of these points are fixed (B and $2^{m/2}B$), but two are obtained dynamically. Using 4-bit windows would then imply storing twice as many points on the stack (8 points for the A window, and 8 other points for the R window). Since stack space is usually a scarce resource on embedded systems, we leveraged the fact that a variable-time algorithm does need to follow a strict schedule of point additions; instead, we can use a non-adjacent form algorithm such that stored windows only need points jA and jR for odd values of j. Indeed, if the window algorithm would call for adding point $6A$ at some step in the sequence of point doublings, we can add point $3A$ one doubling earlier. This trick allows for twice smaller windows, thereby reducing stack usage to the same amount as in the constant-time algorithm.

Use of NAF also lowers the average number of point additions (even with a nominally 4-bit window, additions are applied on average once every *five* doublings), and we could make the windows for B and $2^{m/2}B$ larger, since these use only comparatively cheaper ROM/Flash.

Cofactor. As explained in section 1, the signature verification equation is a linear combination of four curve points (B, $2^{m/2}B$, A and R, with B being the conventional generator, A the public key, and R a point provided as part of the signature); once this linear combination has been computed, it must still be multiplied by the cofactor h. Multiplication by the cofactor is not strictly necessary as per the usual definition of the security of a signature algorithm, but not applying it can induce discrepancies between verifiers about the status of some nominally invalid signature values. Suppose that multiplication by the cofactor h is not used; the usual signature verification is:

$$sB - kA = R$$

while our modified verification equation is:

$$e_0B + e_1 2^{m/2}B - (\delta k \bmod n)A - \delta R = \mathbb{O}$$

If there is on the curve a non-trivial point T of order h, then replacing R with $R + T$ will make the standard verification equation fail, but the second one will still accept the signature if it so happens that the value δ (obtained from the lattice basis reduction algorithm) turns out to be a multiple of h.

Edwards curves, as used in EdDSA, always have a cofactor h which is a multiple of 4 (for Ed25519, the cofactor is $h = 8$). A consequence is that an EdDSA signature verification implementation that uses our improved lattice basis reduction algorithm cannot guarantee that a seemingly valid signature would be deemed acceptable by any other standard-compliant implementation. Signature values that exhibit such discrepancies are nominally invalid and cannot be produced by a valid signature generator; moreover, the security model of signatures is still maintained: that kind of signature may exist only if the private key owner really computed a signature over the same message. However, for protocols that require full agreement of all parties about which signature values are valid or not (e.g. consensus protocols in some distributed systems), it is recommended to apply point normalization, i.e. multiplying by the cofactor h systematically.

On Curve9767, this is a non-issue: Curve9767 has a *prime order*, which means that the cofactor is $h = 1$, and these point normalization issues simply go away. This is one of the reasons why prime order curves are, in all generality, preferable. It is possible to build a prime-order group, amenable to Schnorr signatures, over any Edwards curves with cofactor 4 or 8; this is known as the Ristretto construction[2] and it has relatively small computational overhead over the underlying Edwards curve.

Benchmarks. Our implementation of Curve9767 has been benchmarked over three systems:

– ARM Cortex-M0+: SAM D20 Xplained Pro board (ATSAMD20J18 microcontroller), clocked at 8 MHz, zero wait state for Flash access. Compiler is GCC-7.3.1, with flags: `-Os -mthumb -mlong-calls -mcpu=cortex-m0plus`

– ARM Cortex-M4: STM32F4 "discovery" board (STM32F407VG-DISC1), clocked at 24 MHz, zero wait state (I-cache and D-cache are enabled, but disabling them does not change timings). Compiler is GCC-7.3.1, with flags:
  `-Os -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-s16`
– x86 (64-bit) with AVX2: Intel i5-8259U (Coffee Lake) at 2.3 GHz (TurboBoost is disabled). Compiler is Clang-9.0.0, with flags:
  `-O3 -mavx2 -mlzcnt`

All timings below are in clock cycles. For the ARM implementations, reported times include all opcodes that are part of the function, including the final "ret" (opcodes "`bx lr`" or "`pop { pc }`"); however, the costs of the call opcode itself (`bl`) and the push/removal of function arguments are not accounted as part of the reported costs. For x86 implementations, the cycle counter has been used (`lfence` memory barrier, followed by the `rdtsc` opcode), but the "exact" cost of a function is ill-defined on a system with a deep pipeline with out-of-order execution; reported execution time is the best observed over several executions, with some "warm-up" calls to ensure relevant caches are filled. For functions with a variable execution time, randomized inputs have been used over a few hundred cycles; these values are indicated below with the "≈" sign.

Operation	**Cortex M0+**	**Cortex M4**	**x86+AVX2**
Lattice basis reduction (vartime)	≈106300	≈92010	≈15250
ECDH: key pair generation	1937792	887520	172660
ECDH: compute shared secret	4598756	2054792	392714
Schnorr signature: generate	2054110	965850	182892
Schnorr signature: verify	5688642	2565404	543176
Schnorr signature: verify (vartime)	≈3818000	≈1779000	≈380500

The table above shows benchmarks not only for the signature verification itself, but also for some other operations which can serve as a basis for comparison:

– ECDH, key pair generation: computation of kB, for a secret scalar k and the conventional generator B.
– ECDH, shared secret computation: computation of kQ, for a secret scalar k and a dynamically received point Q.
– Schnorr signature generation: computation of rB, for a secret scalar r and the conventional generator B, and a few extra operations (hashing, and one multiplication and one addition modulo the curve order n).

The measured times also include decoding of received points, which involves point decompression (reconstruction of the y coordinate of the point, using the curve equation and a square root extraction), but this operation is very fast with Curve9767, because its base field allows for efficient square root routines.

In the case of the ARM Cortex M0+, the savings obtained from the improved signature verification algorithm split roughly as follows:

– Halving the number of doublings: about 1.67 million cycles saved.
– Reduction of the number of point additions through NAF: about 302000 cycles saved.
– Lattice basis reduction overhead: about 106300 cycles.

Exactly how much time is saved by halving the number of doublings depends on the used curve, hardware architecture and implementation strategy. For Curve9767, overall savings are about 30 to 33%, compared with the constant-time verification implementation. On a faster curve, the savings may be comparatively lower. On Ed25519, the fastest reported signature verification times[9] are about 110000 cycles; even if only (for instance) 25% of the cost is avoided by the reduction in the number of point doublings, the cost savings would exceed the overhead due to the lattice basis reduction algorithm, since the latter uses only an average of 15250 cycles. Thus, even on that very fast curve, our improved algorithm should make the application of the Antipa *et al* method worthwhile.

4 Conclusion

We presented here a novel variant of Lagrange's algorithm, which allows non-negligible savings in the implementation of Schnorr signature verification. As described by Antipa *et al*[1], the method also applies to ECDSA, albeit with some extra complications due to the fact that the first half of an ECDSA signature is not the complete curve point R, but only the x coordinate of that point (and furthermore reduced modulo the curve subgroup order).

An open question is how fast a constant-time implementation of the lattice basis reduction algorithm could be. Conditional exchanges, shifts and additions and subtractions can be turned into constant-time code with some overhead; moreover, the total number of iterations in a constant-time implementation must match the worst possible case, and our current analysis yields a worst case which is likely to be wildly exaggerated (more than a thousand iterations, while the worst *observed* case was 135 iterations). More analysis is needed to tighten this bound.

An application of our method to other curves, in particular Ed25519, should be produced and benchmarked.

Acknowledgements

We thank Isis Lovecruft for useful discussions about, and review of drafts of, this article.

References

1. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik and S. Vanstone, *Accelerated Verification of ECDSA signatures*, Selected Areas in Cryptography - SAC 2005, Lecture Notes in Computer Science, vol 3897, pp. 307-318, 2005.
2. T. Arcieri, I. Lovecruft and H. de Valence, *The Ristretto Group*, `https://ristretto.group/`
3. D. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering, vol. 2, issue 2, pp. 77-89, 2012.
4. R. Gallant, J. Lambert and S. Vanstone, *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*, Advances in Cryptology - CRYPTO 2001, Lecture Notes in Computer Science, vol. 20139, pp. 190-200, 2001.
5. C. F. Gauss, *Disquisitiones Arithmeticæ*, Springer-Verlag, 1801.
6. S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, `https://tools.ietf.org/html/rfc8032`

7. C. Hermite, *Extraits de lettres de M. Ch. Hermite à M. Jacobi sur différents objets the la théorie des numbres (première lettre)*, Journal für die reine und angewandte Mathematik, vol. 40, pp. 261-277, 1850.
8. J.-L. Lagrange, *Recherches d'arithmétique*, Nouveaux mémoires de l'Académie royale des sciences et belles-lettres de Berlin, pp. 695-795, 1773.
9. I. Lovecruft and H. de Valence, *Dalek cryptography*, `https://dalek.rs/`
10. H. Minkowski, *Geometrie der Zahlen*, Teubner-Verlag, 1896.
11. P. Nguyen and D. Stehlé, *Low-Dimensional Lattice Basis Reduction Revisited*, Algorithmic Number Theory - ANTS 2004, Lecture Notes in Computer Science, vol 3076, pp. 338-357, 2004.
12. T. Pornin, *Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions*, `https://eprint.iacr.org/2020/009`
13. C.-P. Schnorr, *Efficient Identification and Signatures for Smart Cards*, Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science, vol. 435, pp. 239-252, 1989.
14. E. Straus, *Addition chains of vectors (problem 5125)*, American Mathematical Monthly, vol. 70, pp. 806-808, 1964.