EcGFp5: a Specialized Elliptic Curve

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

23 February, 2022

Abstract. We present here the design and implementation of ecGFp5, an elliptic curve meant for a specific compute model in which operations modulo a given 64-bit prime are especially efficient. This model is primarily intended for running operations in a virtual machine that produces and verifies zero-knowledge STARK proofs. We describe here the choice of a secure curve, amenable to safe cryptographic operations such as digital signatures, that maps to such models, while still providing reasonable performance on general purpose computers.

1 EcGFp5 Definition

Let $p = 2^{64} - 2^{32} + 1$. This is a 64-bit prime integer; computations modulo p can be relatively efficiently implemented on a variety of platforms. It has high 2-adicity ($p - 1$ is a multiple of a large power of 2, here 2^{32}) which makes it convenient for STARK proofs[4]. 32-bit integer operations can also be expressed over $GF(p)$ since, for instance, the product of two unsigned 32-bit integers is at most $(2^{32} - 1)^2$, which is lower than p. The Miden VM[13] is an open-source implementation of a virtual machine whose internal opcodes work over elements of $GF(p)$ and can be used to generate and verify STARK proofs over arbitrary program executions. Miden is currently funded by Polygon for blockchain-related purposes, but it can be used in a larger spectrum of situations. Moreover, other projects may want to use the same modulus p, especially but not necessarily in conjunction with zero-knowledge proofs. For the definition of ecGFp5, we consider the abstract and interesting problem of defining a secure prime order group, based on an elliptic curve, amenable to cryptographic operations such as digital signatures, and efficiently implementable in the compute model incarnated by the Miden VM. In that model, all elementary operations in $GF(p)$ have the same unit cost; this includes additions, subtractions, multiplications, and, crucially, divisions.

The chosen curve has the following parameters. We first define the finite field extension $GF(p^5) = GF(p)[z]/(z^5 - 3)$, i.e. the ring of polynomials (in the symbolic variable z) with coefficients in $GF(p)$, and all operations performed modulo the polynomial $z^5 - 3$. Thus, all elements can be represented as five elements of $GF(p)$, corresponding to the five coefficients of a polynomial of degree at most 5. Since $z^5 - 3$ is irreducible over $GF(p)$, this defines a finite field of cardinal p^5.

We define an elliptic curve of equation:

$$y^2 = x(x^2 + 2x + 263z)$$

This is a *double-odd curve*[16]: its order is $2n$ for the 319-bit prime integer:

$$n = 1067993516717146951041484916571792702745057740581727230159139685185762082554198619328292418486241$$

EcGFp5 is then formally defined as the group $\mathbb{G}$ of points of that curve which are *not* points of n-torsion. The neutral element of the group is the point $N = (0, 0)$ (the only point of order 2 on the curve). The sum in the group of two elements P and Q is defined as the curve point $P + Q + N$. As explained in [16], this yields a group with the proper characteristics for defining cryptographic operations such as digital signatures or key exchange:

- The group $\mathbb{G}$ has prime order n.
- Elements of $\mathbb{G}$ can be uniquely encoded into a field element; the decoding process is unambiguous and inherently verifies that the provided encoding was valid and canonical.
- Group operations can be computed with efficient complete formulas.

Several systems of coordinates can be used. In general, it is recommended to use (x, u) coordinates, in which $u = x/y$ for element (x, y) (for the neutral N, we use $u = 0$). If both x and u are expressed as fractions (denoted X/Z and U/T, respectively), then general point addition formulas have a cost of 10 multiplications in the field (denoted 10M), and specialized formulas for sequences of doublings have a per-doubling cost of 2M+5S (two multiplications and five squarings in the field). As will be detailed in the next sections, though, within the target compute model, it is in fact more efficient to switch to affine Weierstraß coordinates and formulas.

In the next sections, we will:

- justify the choice of a degree-5 field extension;
- describe the implementation of field and curve operations in the target compute model (thereafter called "in-VM");
- formalize the choice criteria for the curve parameters;
- provide some details on the implementation of the curve when not working in the VM (i.e. the "out-of-VM" situation).

A copy of this paper, a test implementation in Python that emulates the VM model to measure costs, and a reference implementation in Rust, are provided at:

`https://github.com/pornin/ecgfp5/`

2 Choice of Field

Since p is a 64-bit integer, we need to work in a field extension $GF(p^k)$ in order to have a field large enough to obtain a curve with adequate security. We aim at the usual "128-bit security" level. For such a level, we need a field with at least a 256-bit order, hence $k \geq 4$. Robustness of elliptic curve discrete logarithm in extension fields has been studied in various articles. A rough summary is the following:

- If the extension degree k is composite, then Weil descent attacks may apply, using a tower of field extensions to turn the problem into a discrete logarithm in an higher genus curve on a smaller field[9,3]. To avoid such issues, a prime degree is highly recommended. Diem showed that if k is prime and not lower than 11, then such attacks cannot work[6].
- A related attack using Gröbner bases was described by Gaudry[8]; its complexity was further analyzed by Joux and Vitse, along with some possible variants[12].

For performance reasons, we would like to have k as small as possible. Using $k = 4$ would allow the known attacks on quartic extension fields[3], with complexity about $O(p^{3/2}) \approx 2^{96}$. Though this value is quite larger than what can practically be implemented, it still falls short of the expected "128-bit" level. Thus, we need at least $k = 5$.

With $k = 5$, Gaudry's attack entails computing about $p^{2-2/5} \approx 2^{102.4}$ systems of polynomial equations, and obtaining a Gröbner basis for each of them. Each system would contain 5 equations with 5 unknowns, and a total degree $2^{k-1} = 16$; it is expected that obtaining the basis will require using the FGLM algorithm[7] with complexity $O(kD^3)$, with k the number of unknowns (here, $k = 5$) and D the degree of the underlying ideal, which should be close to $2^{k(k-1)} = 2^{20}$. The involved matrix should be mostly empty and a lower complexity might be achieved, but even with very optimistic assumptions, it is unlikely to go below $O(D^2) \approx 2^{40}$. This leads to a total theoretical complexity of at least 2^{142}, well beyond the target 128-bit level. Joux and Vitse's variant has cost $O(Cp^2)$ for some constant C that depends on k, again above the 128-bit level.

We can thus claim that a degree-5 extension field, $GF(p^5)$, is sufficient to achieve 128-bit security.

All finite fields with the same cardinal are isomorphic to each other; we can thus choose whatever definition of that field provides the best performance. Field extensions of degree k are classically defined as the quotient of the ring of polynomials in the base field, by a given unitary irreducible polynomial M of degree k. Since multiplications in the extension field will involve reductions modulo M, performance should be best if using an M of minimal Hamming weight, and with non-zero coefficients as close to 1 or -1 as possible; moreover, a modulus with format $z^5 - c$ for some constant c makes the Frobenius operator especially inexpensive. Among polynomials in $GF(p)[z]$, none of the following polynomials happens to be irreducible: z^5, $z^5 \pm 1$, $z^5 \pm z^i \pm 1$ for any $i \in [1; 4]$, $z^5 \pm 2$. The next best choices are $z^5 - 3$ and $z^5 + 3$, which are both irreducible; we thus choose $M = z^5 - 3$.

3 In-VM Implementation

3.1 VM Opcodes

We assume here that the following opcodes are offered by the target compute model, and all have cost exactly 1 cycle:

- `add`, `sub`, `mul` and `div` perform respectively addition, subtraction, multiplication and division in $GF(p)$. Division fails if the divisor is zero; it is up to the caller to make sure that this situation does not happen. Negation has a specific opcode (`neg`) but could also be implemented with a subtraction from zero.
- `and`, `or`, `xor` and `not` perform operations on Boolean values. A "true" is represented as the $GF(p)$ element 1, while a "false" is 0. Any other value triggers a failure. Opcodes `eq` and `neq` compare two $GF(p)$ elements and return such Boolean value if the two operands, are, respectively, equal to each other, or different from each other.
- `select` applied on three values x, y and c returns x if $c = 0$, or y if $c = 1$. The control value c must have a Boolean 0-or-1 value.
- `add32`, `sub32`, `mul32`, `div32`, `shl32`, `shr32`, and `gte32` implement operations on 32-bit values (for addition, subtraction, multiplication, division, shift left, shift right,

and greater-or-equal comparisons, respectively). Addition and subtraction are computed modulo 2^{32} and have carry/borrow support for both input and output. Multiplication yields a 64-bit output (which always fits in a $GF(p)$ element). Shifts and comparisons operate on unsigned values; the left shift truncates its output to 32 bits. Moreover, shift counts must be fixed constants. All 32-bit opcodes assume that the operands are in the proper range (0 to $2^{32} - 1$).

The names above do not exactly match the names used by the Miden VM assembly specification[14]; for instance, what we call here `add32` is known as `u32addc.unsafe` in the specification document.

The compute model should be understood as a general circuit emulation in which only arithmetic gates have a cost, while data routing is free, provided that it can be resolved statically. In the context of a VM executing a program consisting of opcodes, this means that function calls, loop control, reading from memory and writing to memory are all free (their cost is zero); *however*, this does not extend to data-dependent conditional jumps, and array accesses at data-dependent indexes. These operations are possible in Miden but very expensive; in the context of this paper, we simply consider them to be forbidden. In a sense, we use a compute model which is close to constant-time implementations, although for different reasons.

3.2 Field Operations

An element x of $GF(p^5)$ is represented as five coefficients x_0 to x_4, such that $x = x_0 + x_1 z + x_2 z^2 + x_3 z^3 + x_4 z^4$. Addition and subtraction are simply done coefficient-wise; thus, an addition in $GF(p^5)$ boils down to five `add` opcodes, for a cost of 5.

Multiplication. Multiplication in $GF(p^5)$ ($d \leftarrow a + b$) can be done in a straightforward way:

$$
\begin{aligned}
d_0 &\leftarrow a_0 b_0 + 3(a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1) \\
d_1 &\leftarrow a_0 b_1 + a_1 b_0 + 3(a_2 b_4 + a_3 b_3 + a_4 b_2) \\
d_2 &\leftarrow a_0 b_2 + a_1 b_1 + a_2 b_0 + 3(a_3 b_4 + a_4 b_3) \\
d_3 &\leftarrow a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 + 3 a_4 b_4 \\
d_4 &\leftarrow a_0 b_4 + a_1 b_3 + a_2 b_2 + a_3 b_1 + a_4 b_0
\end{aligned}
$$

The multiplications by 3 come from the reduction modulo the polynomial $z^5 - 3$. Any constant in $GF(p)$ other than 3 would yield the same overall cost, except 0, 1 or -1, which would be cheaper; however, polynomials z^5, $z^5 + 1$ and $z^5 - 1$ are not irreducible, and do not yield a proper field.

Overall multiplication cost is 49. Other techniques such as Karatsuba or Toom-Cook may reduce the number of multiplications, but at the cost of a higher number of additions and subtractions; in our compute model, these do not seem to provide overall cost reductions.

Squaring can be done with lower cost since, for instance, product $a_0 b_1$ and $a_1 b_0$ yield the same value when $a = b$. The cost of a squaring operation is then 34.

Inversion. Inversion in $GF(p^5)$ can be computed very efficiently thanks to a method first described by Itoh and Tsujii[11]. Define the integer $r = 1 + p + p^2 + p^3 + p^4$. The following holds:

$$p^5 - 1 = (p - 1)r$$

Therefore, for any non-zero x in $GF(p^5)$, we have:

$$x^{p^5-1} = 1 = (x^r)^{p-1}$$

Thus, x^r is a root of the polynomial $X^{p-1} - 1$. Since the roots of that polynomial are exactly the elements of $GF(p)$, this implies that $x^r \in GF(p)$ for any x in $GF(p^5)$. We can thus write the inverse of x as:

$$\frac{1}{x} = \frac{x^{r-1}}{x^r}$$

which can be computed as the product of x^{r-1} (an element of $GF(p^5)$) by the inverse of x^r (an element of $GF(p)$).

Values x^{r-1} and x^r can furthermore be computed with appropriate use of the Frobenius operator. Define $\phi_1(x) = x^p$ for any $x \in GF(p^5)$; this is a field automorphism, i.e. $\phi_1(x+y) = \phi_1(x) + \phi_1(y)$, and $\phi_1(xy) = \phi_1(x)\phi_1(y)$ for any x and y. Thus, if $x = \sum_i x_i z^i$, then:

$$\begin{aligned}\phi_1(x) &= \sum_{i=0}^{4} \phi_1(x_i) z^{pi} \\ &= \sum_{i=0}^{4} x_i (3^{i\lfloor p/5 \rfloor}) z^{i(p \bmod 5)}\end{aligned}$$

In our case, $p \bmod 5 = 1$, so the Frobenius operator is computed by simply multiplying all five coefficients by precomputed constants, the first of which being furthermore equal to 1. This is done in cost 4. We similarly define $\phi_2(x) = \phi_1(\phi_1(x))$, which is also computed in cost 4.

With the Frobenius operator, we compute x^{r-1} as:

$$\begin{aligned}x^{r-1} &= x^{p+p^2+p^3+p^4} \\ &= \phi_1(x)\phi_1(\phi_1(x))\phi_2(\phi_1(x)\phi_1(\phi_1(x)))\end{aligned}$$

which entails three Frobenius operators and two multiplications in $GF(p^5)$. Once x^{r-1} is obtained, x^r is computed by multiplying that value with x; since x^r is known to be in $GF(p)$, we only need to compute the first coefficient, with cost 10.

Inversion in $GF(p)$ has cost 1 (this is a single `div` opcode); however, we have to add a small corrective action to avoid a division by zero, in case the input x was equal to zero: before inverting $y = x^r$, we compare it with zero (`eq` opcode), then add the Boolean result (a $GF(p)$ element with value 0 or 1) to y. Thus, if $x = 0$, we end up inverting $y' = 1$ instead of $y = 0$, and the division opcode does not fail. This corrective action has cost 2.

Final multiplication of x^{r-1} by the resulting x^{-r} is done in five `mul` opcodes. The overall cost of inversion in $GF(p^5)$ is 128 cycles (it would be 126 if the `div` opcode tolerated a divisor equal to 0). Note that if $x = 0$, the inversion process described above does not fail; instead, it returns 0. This is a feature; it simplifies some operations later on.

General division is the combination of a multiplication and an inversion, with cost 177.

Legendre Symbol. We define the Legendre symbol of x as the value $x^{(p^5-1)/2}$; it is equal to 0 if $x = 0$, to 1 if x is a non-zero quadratic residue, or -1 if x is not a square in $GF(p^5)$. Again using the value $r = 1 + p + p^2 + p^3 + p^4$, we find that:

$$x^{(p^5-1)/2} = (x^r)^{(p-1)/2}$$

Thus, the Legendre symbol of $x \in GF(p^5)$ is equal to the Legendre symbol of x^r in $GF(p)$. As in the case of inversion, the computation of x^r is efficient. In $GF(p)$, we note that $(p-1)/2 = 2^{63} - 2^{31}$, leading to the following process:

1. $y \leftarrow x^r$ (result is in $GF(p)$)
2. $y_{31} \leftarrow y^{2^{31}}$ (with 31 `mul` opcodes)
3. $y_{63} \leftarrow y_{31}^{2^{32}}$ (with 32 `mul` opcodes)
4. $t \leftarrow y_{63}/y_{31}$ (one `div` opcode)

The overall cost is 186.

Square Root. We can use, again, the Frobenius operator to speed up square roots, by noticing that, for $x \neq 0$ in $GF(p^5)$:

$$\begin{aligned}\sqrt{x} &= \sqrt{\frac{x^r}{x^{r-1}}} \\ &= \frac{\sqrt{x^r}}{x^{(r-1)/2}}\end{aligned}$$

since $r - 1 = p + p^2 + p^3 + p^4$, which is an even integer. We can compute $x^{(r-1)/2}$ as:

$$\begin{aligned}x^{(r-1)/2} &= x^{p(1+p^2)(p+1)/2} \\ &= \phi_1(x^{(p+1)/2}\phi_2(x^{(p+1)/2}))\end{aligned}$$

We can write $x^{(p+1)/2} = x^{2^{63}+1}/x^{2^{31}}$, allowing the computation of that value with 63 squarings, one multiplication and one inversion in $GF(p^5)$. Once we obtained $x^{(p+1)/2}$, we use it to compute $x^{(r-1)/2}$ as show above, with two Frobenius operators and one multiplication. We can furthermore derive x^r from $x^{(r-1)/2}$ with a squaring (to get x^{r-1}) then a multiplication by x; the latter only needs to compute the lowest coefficient, since $x^r \in GF(p)$.

At that point, we have to compute the square root of $y = x^r$ in $GF(p)$. Moduli with high 2-arity are known to be inconvenient for computing square roots. We use the following process, which really is the Tonelli-Shanks algorithm, specialized to our compute model in which data-dependent conditional jumps are forbidden, but multiplications are inexpensive:

1. Let $n = 32$ and $q = 2^{32} - 1$, so that q is odd and $p = q2^n + 1$. Let g be a primitive 2^n root of unity in $GF(p)$ (we can use $g = 7^q \bmod p$, since 7 is a non-QR in $GF(p)$). We precompute values $g_i = g^{2^i}$ for $i = 0$ to $n - 1$.
2. $(u, v) \leftarrow (y^{(q+1)/2}, y^q)$
3. For $i = n - 1$ down to 1:
   (a) $w \leftarrow v^{2^{i-1}}$ (with $i - 1$ squarings)

(b) If $w = -1 \bmod p$, then: $(u, v) \leftarrow (ug_{n-i-1}, vg_{n-i})$ (the new (u, v) are always computed, but kept with `select` opcodes only if an `eq` opcode declares that the computed w is indeed equal to -1)

4. If $v = 0$ or 1, then y was indeed a square, and u contains one of its square roots. Otherwise, y was not a square, and thus x was not a square either; in that case, we arrange to set v to zero (e.g. with an extra `select`).

The algorithm above entails $(n-1)(n-2)/2 = 450$ squarings, and some other operations, for a total of 659 cycles for the square root in $GF(p)$. Combined with the computations of $x^{(r-1)/2}$ and x^r, and finally combining together the value, we compute a square root in $GF(p^5)$ in a total of 3261 cycles. The routine returns two values, the square root itself, and a Boolean value reporting the success of the process; if the input value was not a square, the returned "square root" is zero, and the Boolean is zero.

Cost Summary. We obtain the following costs for in-VM operations in $GF(p^5)$:

Operation	Cost (cycles)
addition	5
subtraction	5
multiplication	49
squaring	34
inversion	128
division	177
Legendre symbol	186
Square root	3261

An important point here is that inversions in $GF(p^5)$ are quite inexpensive: the cost of an inversion is only about 2.57 times the cost of a multiplication. This is not the usual situation when dealing with elliptic curve implementations; it impacts the strategy we will use, in particular the point addition formulas.

3.3 Curve Formulas

Since inversions in $GF(p^5)$ are quite efficient (their cost is lower than three times the cost of a multiplication in $GF(p^5)$), the most efficient formulas for computing point additions and doublings are obtained by working with the short Weierstraß equation and affine coordinates. Moreover, *any* elliptic curve over $GF(p^5)$ can be expressed as a short Weierstraß curve with a suitable change of variable; thus, no curve type will be any better or worse than any other, efficiency-wise, for in-VM computations.

Change of Variable. A double-odd curve such as ecGFp5 has equation $y^2 = x(x^2 + ax + b)$ for two constants a and b (for ecGFp5, $a = 2$ and $b = 263z$). It can be converted to the short Weierstraß equation:

$$Y^2 = X^3 + AX + B$$

with constants:

$$A = (3b - a^2)/3$$
$$B = a(2a^2 - 9b)/27$$

using the following change of variable:

$$(X, Y) = (x + \frac{a}{3}, y)$$

This change of variable is very inexpensive: it suffices to add a single constant to the x coordinate. Moreover, that constant is in $GF(p)$ for ecGFp5, leading to an addition in a single cycle in the VM.

EcGFp5, however, is not *exactly* the elliptic curve with equation $y^2 = x(x^2+ax+b)$, but a subset thereof, consisting of (exactly) the points which are not the double of any other point. In order to perform computations on the short Weierstraß curve, we also need to map into the subgroup of points of n-torsion on the short Weierstraß curve, which entails adding the point N. This can be done while still on the original equation, since $(x, y) + N = (b/x, -by/x^2)$. Another method, which is even simpler, is to combine the addition of N with the decoding process. Given an encoded point w (nominally equal to y/x for the ecGFp5 element (x, y)), compute the following:

1. $e \leftarrow w^2 - a$
2. $\Delta \leftarrow e^2 - 4b$
3. $(x_1, x_2) \leftarrow ((e + \sqrt{\Delta})/2, (e - \sqrt{\Delta})/2)$ (if Δ is not a square, then either $w = 0$, in which case the point is N and should be decoded as the point-at-infinity in the short Weierstraß curve; or $w \neq 0$, and there is no solution, w is not a validly encoded point)
4. If x_1 is a quadratic residue, then set $x = x_1$; otherwise, set $x = x_2$.
5. Return $(X, Y) = (x + a/3, -wx)$

Step 4 is where processing diverges from normal decoding in double-odd curves, where we would have chosen the x value which is *not* a quadratic residue instead. This decoding process works because a given $w = y/x$ value is shared by two points on the curve, $P + N$ (which is in ecGFp5) and $-P$ (which is in the subgroup of points of n-torsion); here, we simply use the latter, and take the negation into account by computing $y = -wx$ instead of $y = wx$.

It shall be noted that if $w = 0$, then the decoding above computes $\Delta = a^2 - 4b$, which is not a quadratic residue. Moreover, in that case, the "point-at-infinity" should be returned, and that point does not have defined (X, Y) coordinates. In a practical implementation, a point in affine coordinates on the short Weierstraß curve really is a set of *three* values: the coordinates X and Y, which are in $GF(p^5)$, and a Boolean flag I which, when non-zero, indicates that the point is the point-at-infinity, and the values of X and Y should be ignored.

Point Addition. The sum of two points (X_1, Y_1) and (X_2, Y_2) is the point (X_3, Y_3) with:

$$\begin{aligned}
\lambda &= \frac{Y_2 - Y_1}{X_2 - X_1} \\
X_3 &= \lambda^2 - X_1 - X_2 \\
Y_3 &= \lambda(X_1 - X_3) - Y_1
\end{aligned}$$

Famously, these formulas are not complete; if $X_1 = X_2$ then either $Y_1 = -Y_2$, in which case the sum is the point-at-infinity (sum of a point and its opposite); or $Y_1 = Y_2$, which means

that the point is added to itself, and λ must instead be computed as:

$$\lambda = \frac{3X_1^2 + A}{2Y_1}$$

A *complete routine* that supports all cases looks like this:

1. Inputs are point (X_1, Y_1) (with infinity flag I_1) and (X_2, Y_2) (with flag I_2).
2. Compare X_1 with X_2, yielding a Boolean s_x with value 1 if they are equal, 0 otherwise. This entails five `eq` opcodes, and four `and` opcodes, for a cost of 9. Similarly, compare Y_1 with Y_2, yielding s_y.
3. Set λ_0 to $Y_2 - Y_1$ if $s_x = 1$, or to $3X_1^2 + A$ if $s_x = 0$ (both values are computed, and `select` opcodes are used to keep the right one, depending on the value of s_x).
4. Set λ_1 to $X_2 - X_1$ if $s_x = 1$, or to $2Y_1$ if $s_x = 0$.
5. $\lambda \leftarrow \lambda_0 / \lambda_1$
6. $X_3 \leftarrow \lambda^2 - X_1 - X_2$
7. $Y_3 \leftarrow \lambda(X_1 - X_3) - Y_1$
8. $I_3 \leftarrow s_x$ AND s_y
9. If $I_1 \neq 0$, then replace (X_3, Y_3, I_3) with (X_2, Y_2, I_2)
10. If $I_2 \neq 0$, then replace (X_3, Y_3, I_3) with (X_1, Y_1, I_1)

This routine handles all edge cases (point doublings, point-at-infinity as input or output operand) with a fixed cost of 387 cycles in the VM. This cost is about 7.9 times the cost of a single multiplication: this is faster than the best know curve point adding formulas that do not involve any inversion.

Optimizations are possible in some cases:

- When it is *a priori* known that the addition is not an edge case, then we can avoid in particular the squaring involved in computing λ_0 for a point doubling, and all the `select` opcodes, leading to a routine with cost 290.
- For explicit point doublings, the `select` opcodes can also be skipped, since the double of a non-infinity n-torsion point cannot be the point-at-infinity; we can simply keep the infinity flag unchanged, and apply the doubling formulas on the coordinates. This leads a point doubling in 326 cycles.

Point Multiplication. In the specific case of the multiplication of a point P by a scalar v, the following process can be used:

1. Reduce the scalar v modulo n, then add n to get a value k in the n to $2n - 1$ range.
2. Split the scalar into chunks $\lceil 321/w \rceil$ of w bits, for a given window width w (experimentally, $w = 4$ seems to be the best choice here). Each chunk c_i yields a signed digit d_i with the following:
   (a) Initialize a carry m to 0.
   (b) For each chunk c_i (in least-to-most significant order), add m to the value of c_i. If $m + c_i \leq 2^{w-1}$, then set d_i to that value, and set m to zero; otherwise, set d_i to $m + c_i - 2^w$, and set m to 1.

All digits are between $-2^{w-1}+1$ and $+2^{w-1}$; moreover, with the chosen parameters, it can be shown that the top digit is necessarily greater than or equal to 1. Since the VM works with *unsigned* 32-bit integers (mapped to $GF(p)$ elements), the sign and absolute value of each digit may be returned separately.

3. Fill an array W with points iP for $i = 1$ to 2^{w-1}. This is called the "window". For an index value d between -2^{w-1} and $+2^{w-1}$, the point dP can be recovered with a lookup process (detailed below).
4. For all digits d_i, starting with the second-to-top digit down to d_0:
   (a) Multiply Q by 2^w with w successive doublings.
   (b) Lookup point d_iP from the window, and add it to Q.

It can be shown that in this process, if input P was not the point-at-infinity, then for all digits except the last two (d_1 and d_0), the addition of the looked-up point d_iP to the current Q cannot be an edge case of the point addition formulas; thus, each of these additions can be the specialized routine with cost 290; only the last two iterations need to use the generic routine with cost 387.

All these operations, including the building of the window, can be performed assuming that the input P is not the point-at-infinity; it suffices to combine (with a Boolean OR) the initial flag I_P with the current flag I_Q to obtain a proper result even in case the input P is the point-at-infinity. Note that window building can then use the specialized point addition, and the infinity flag of each window element needs not be stored.

The window lookup for digit value d is done as follows:

1. Initialize variables X and Y to copies of X_P and Y_P.
2. For indices $i = 2$ to 2^{w-1}, replace X and Y with the coordinates of iP (from the window) if and only if $|d| = i$.
3. If $d < 0$, then replace Y with $-Y$.
4. Set the flag I to 1 if $d = 0$, or to 0 otherwise.

Lookup cost increases with window size, with an overhead of 11 cycles per extra point.

Overall cost of the point multiplication function, including the addition of n to v and the split into digits, and the building of the window, was measured to be 138482 cycles.

Key Pair and Signature Generation. A special case of point multiplication is when the point to multiply is the conventional generator point G. This is the main curve operation in public/private key pair generation, as well as signature generation. In that case:

- The window can be precomputed.
- Since there is no cost for window building, a larger window may offer better performance, although lookup costs will dominate with large windows.
- Several windows for precomputed multiples of G may be used conjointly, in order to reduce the number of loop iterations and doubles. For instance, with 8 windows for all $2^{40i}G$ (with $i = 0$ to 7), 7/8th of the point doublings can be avoided, leading to a considerable speed-up.

There are many possible trade-offs between window size, number of windows, and code size. When generating or verifying STARK proofs, the whole implementation can be conceptually unrolled, and large tables of constants used; other situations might call for more compact implementations.

Signature Verification. Verification of a Schnorr signature entails checking that $dG + eQ = R$ for some scalars d and e (obtained from the signature itself, and some hashing), conventional generator G, public key Q, and the point R whose encoding is the first half of the signature. In general, there are many optimizations that can be applied on signature verification:

- The size of the two scalars can be about halved by using the Antipa *et al* method[1], usually with Lagrange's algorithm for lattice reduction[15].
- Scalar representation as signed digits can use the w-NAF representation, in which most digits are zero and all the non-zero digits are odd, leading to fewer and faster window lookups.
- Several verifications can be performed simultaneously with a randomized batch verification process, allowing important cost sharings.

Unfortunately, most of these optimizations require conditional execution of some kind, depending on the involved data. This is usually not a problem (signature verification nominally happens only over public data), but in the VM compute model, conditional execution is quite inconvenient.

Computation of $dP + eQ$ for two points P and Q, and two scalars d and e, can still use Straus's algorithm[18] (often known in cryptography as "Shamir's trick") to mutualize point doublings, i.e. perform about 320 doublings in total, instead of 640 as would be obtained with two seperate point multiplication operations.

4 Curve Parameters Selection

As we saw in section 3.3, in-VM curve operations will preferably use a short Weierstraß equation and affine formulas. *Any* elliptic curve is amenable to such a representation; moreover, the specific values of the Weierstraß constants A and B have little to no incidence on in-VM performance: A is only used in an addition over $GF(p^5)$ as part of the doubling formulas, and B is not used at all in the formulas. Thus, the in-VM compute model does not imply any constraint on the curve equation type we will use. We are free to select a curve type that favours out-of-VM performance, as long as it provides the required security characteristics.

For proper security in arbitrary protocols, we need a prime-order group with a unique, canonical and verifiable encoding[5]. In practice, this restricts our choice to the following:

- A curve with a prime order. This requires using the generic short Weierstraß equation. Encoding output consists of the x coordinate, along with a single "sign" bit for y to designate which square root of y^2 is intended.
- A double-odd curve[16], with order $2n$ for a prime n. An element of the prime order group is encoded as a single field element.
- A Montgomery or twisted Edwards curve, of order $4n$ or $8n$ for a prime n, along with the Decaf/Ristretto encoding process[10,2].

All three kinds offer division-less complete formulas for safe out-of-VM processing. The formulas for prime-order short Weierstraß curves[17] are somewhat slower than for the other two (12M for general addition, 8M+3S for doubling). Twisted Edwards curves have the fastest general addition formulas (8M), and point doubling with cost 4M+4S (an alternate choice

of representation called "inverted coordinates" offers addition in 9M+1S and doubling in 3M+4S). However, double-odd curves have better doubling formulas (2M+5S per-doubling overhead); the encoding/decoding process of double-odd curves is also somewhat simpler than that of Decaf/Ristretto, and allows for efficient validation that a point is decodable and canonical (with only a Legendre symbol). We will thus choose a double-odd curve.

There are several representations for double-odd curves; in general, it is recommended to use fractional (x, u) coordinates, for which complete formulas are known. A point is represented as a quadruplet $(X{:}Z{:}U{:}T)$, which is such that $x = X/Z$ and $u = x/y = U/T$. Point addition formulas on that representation entail some multiplications by the equation constants a and b, and also the constants $\alpha = (4b - a^2)/(2b - a)$ and $\beta = (a - 2)/(2b - a)$. Choosing $a = 2$ implies that $\alpha = 2$ and $\beta = 0$, which is convenient.

If $a = 2$, which is an element of $GF(p)$ (the base field), we must choose b outside of $GF(p)$; otherwise, the curve over the base field would be a subgroup of the curve over $GF(p^5)$, and it would not be possible to obtain total curve order $2n$ for a prime n. To speed up multiplications by b, we will still want to use a constant b equal to $b_i z^i$ for some $i \in [1; 4]$, and b_i as small as possible as an integer (in absolute value). We thus use the following search process:

1. $c \leftarrow 1$
2. For $i = 1$ to 4:
   (a) If curve $y^2 = x(x^2 + 2x + cz^i)$ has order $2n$ with n prime, then return $b = cz^i$.
   (b) If curve $y^2 = x(x^2 + 2x - cz^i)$ has order $2n$ with n prime, then return $b = -cz^i$.
3. $c \leftarrow c + 1$
4. Loop to step 2.

As explained in [16], a curve of equation $y^2 = x(x^2 + ax + b)$ may be double-odd (i.e. order $2n$ for an odd integer n) only if neither b nor $a^2 - 4b$ is a quadratic residue; this gives a fast test that allows skipping most of the expensive point counting operations in the process described above.

Using the process above, the first usable curve is obtained for $b = 263z$. This yields the curve whose parameters were given in section 1.

While the curve selection process is not known to induce any bias or select a curve with uncommon properties, we still checked that its embedding degree is large. For a curve defined over a finite field $GF(q)$ and with a subgroup of prime order n (that does not divide q), the embedding degree is the smallest integer $e > 0$ such that n divides $q^e - 1$ (or, said otherwise, e is the multiplicative order of q modulo n). If e is very small, then the Weil, Tate and similar pairings can be computed, reducing the discrete logarithm in the curve into the discrete logarithm over the multiplicative group of invertible elements in $GF(q^e)$. A randomly selected curve should have a very large embedding degree e (about the same size as n); a low embedding degree does not necessarily imply a weakness (except if e is so small that the discrete logarithm in $GF(q^e)$ can be computed more efficiently than in the curve itself), but it would hint at some unexpected internal structure. In the case of ecGFp5, we checked that $e = (n - 1)/5$, i.e. a 317-bit integer, close to the size of n itself, as is expected of a randomly selected curve. To perform this check, notice that e is necessarily a divisor or $n - 1$; thus, we can factor $n - 1$ to try all values $(n - 1)/r$ for any prime r that divides $n - 1$. The factorization of $n - 1$ is:

$$\begin{aligned} n - 1 = {} & 2^5 \cdot 5 \cdot 163 \cdot 769 \cdot 1059871 \\ & \cdot 25324382672016243125485781410012 7 \\ & \cdot 19840052305318400281440353691816272491634384252056 1 \end{aligned}$$

5 Out-of-VM Implementation

EcGFp5 was designed to match the abilities of the target VM compute model, not the abilities of any specific concrete CPU such as modern x86 or ARM. Out-of-VM performance is thus expected to be lower than what is usually expected from fast elliptic curves with 128-bit security, for two reasons:

- Operations in $GF(p)$ have some overhead. The specific value of $p = 2^{64} - 2^{32} + 1$ allows for some fast reduction techniques, but fast reduction is still more expensive than no reduction at all.
- $GF(p^5)$ is a 320-bit field, 1.25 times larger than a 256-bit field. Point multiplication has a cost cubic in the size of the field (with commonly used field sizes); thus, as a very rough approximation, we should expect a cost factor $1.25^3 \approx 1.95$ compared to a usual curve with 128-bit security.

We implemented ecGFp5 in the Rust programming language. The implementation is constant-time and efficient; it is nonetheless fully portable (it uses only the `core` library; it includes no inline assembly, no architecture-specific intrinsics, and no `unsafe` code). We still achieve the following performance on an Intel i5-8259U "Coffee Lake" CPU (using Rust compiler version 1.57.0, with extra flags "-C target-cpu=native" to allow the compiler to use opcodes available on that CPU, in particular `mulx`):

Operation	**Cost (cycles)**
$GF(p)$ addition	4.02
$GF(p)$ subtraction	3.02
$GF(p)$ multiplication	10.18
$GF(p)$ inversion	737.39
$GF(p)$ Legendre symbol	714.20
$GF(p)$ Square root	5430.19
$GF(p^5)$ addition	8.80
$GF(p^5)$ subtraction	5.78
$GF(p^5)$ multiplication	94.03
$GF(p^5)$ squaring	68.63
$GF(p^5)$ inversion	1069.75
$GF(p^5)$ Legendre symbol	1042.20
$GF(p^5)$ Square root	12410.38
ecGFp5 point addition	1328.50
ecGFp5 point doubling	985.37
ecGFp5 point doubling ×5	3971.39
ecGFp5 point multiplication	363168.03
ecGFp5 generator multiplication	109516.20
ecGFp5 mul+add verification	336952.88

In this table, an average per-operation time is reported for a long sequence of dependent operations: the output of each operation is used as part of the input to the next one. When some operations do not depend on each other, they may be run concurrently by the CPU to some extent; this is how, for instance, a subtraction in $GF(p)$ costs 3 cycles, but a subtraction in $GF(p^5)$ can be done in less than 6 cycles instead of 15.

"Point doubling ×5" means five successive point doublings. The fractional (x, u) formulas on double-odd curves include optimizations for long sequences of successive doublings, for a cost of 2M+1S+j(2M+5S) for j doublings; this is why that sequence cost is only about four times the cost of a single doubling, instead of five times.

5.1 Field Operations

There are several possible representations of integers modulo p. In our implementation, we chose to use *Montgomery representation*: an element $x \in GF(p)$ is represented as $2^{64}x \bmod p$, normalized as an integer in the $[0; p-1]$ range. This representation is coupled with *Montgomery multiplication*, which, given x and y in $GF(p)$, computes $xy/2^{64} \bmod p$. Hence, the Montgomery multiplication of $2^{64}x$ and $2^{64}y$ yields $2^{64}xy$, which is the Montgomery representation of the product xy. The cornerstone of this support is the reduction function, which, given a 128-bit input x (lower than $2^{64}p = 2^{128} - 2^{96} + 2^{64}$) returns $x/2^{64} \bmod p$. This reduction is implemented in Rust as follows:

```
const fn montyred(x: u128) -> u64 {
    let xl = x as u64;
    let xh = (x >> 64) as u64;
    let (a, e) = xl.overflowing_add(xl << 32);
    let b = a.wrapping_sub(a >> 32).wrapping_sub(e as u64);
    let (r, c) = xh.overflowing_sub(b);
    r.wrapping_sub(0u32.wrapping_sub(c as u32) as u64)
}
```

This reduction works as follows:

- The input x is split into two 32-bit words and one 64-bit word: $x = x_0 + 2^{32}x_1 + 2^{64}x_2$ with x_0 and x_1 being unsigned 32-bit integers, and x_2 being 64-bit. Note that the assumption on the range of the function input implies that $x_2 < p$.
- The third function line adds $2^{32}x_0$ to $2^{32}x_1 + x_0$, with a carry in e. We thus obtain:

$$a = -2^{64}e + 2^{32}(x_0 + x_1) + x_0$$

- On the fourth line, we compute:

$$\begin{aligned} b &= -2^{64}e + 2^{32}(x_0 + x_1) + x_0 + 2^{32}e - (x_0 + x_1) - e \\ &= 2^{32}(x_0 + x_1) - x_1 - ep \\ &= 2^{32}x_0 + (2^{32} - 1)x_1 - ep \end{aligned}$$

  If $x_0 + x_1 < 2^{32}$, then $e = 0$ and this value is bounded as:

$$0 \le b = 2^{32}x_0 + (2^{32} - 1)x_1 = x_0 + (2^{32} - 1)(x_0 + x_1) \le 2^{32} - 1 + (2^{32} - 1)^2 < p$$

  Otherwise, if $x_0 + x_1 \ge 2^{32}$, then $e = 1$, and:

$$0 = 2^{64} - (2^{32} - 1) - p \le b = 2^{32}(x_0 + x_1) - x_1 - p \le 2^{32}(2^{33} - 2) - 1 - p < p$$

  In both cases, the computed value indeed fits in the variable `b` without truncation, and we know that $b < p$.

– Since $1 = 2^{32} - 2^{64} \mod p$, we know that:

$$\begin{aligned} x_0 + 2^{32}x_1 &= 2^{32}x_0 - 2^{64}x_0 + 2^{64}x_1 - 2^{96}x_1 \mod p \\ &= 2^{64}x_0 - 2^{96}x_0 - 2^{64}x_0 + 2^{64}x_1 - 2^{96}x_1 \mod p \\ &= -2^{64}(2^{32}(x_0 + x_1) - x_1) \mod p \\ &= -2^{64}b \mod p \end{aligned}$$

  Therefore, $x/2^{64} = x_2 - b \mod p$. At that point, we have both x_2 (in `xh`) and b (in `b`), and both are in the $[0; p-1]$ range; thus, we only have to do a single subtraction, and potentially adding back p if this subtraction yields a negative result. This is what is done in the last two lines of the function. Specifically, a subtraction is done with result in `r` and borrow flag in `c`. If the borrow is 1, then we subtract it from zero with a 32-bit wrapping operation, which then yields $2^{32} - 1 = -p \mod 2^{64}$. We then subtract that value from `r`, which is equivalent to adding p (modulo 2^{64}). It is easily seen that if there were no borrow (`c` is zero), then the last line does not change the value of `r`. In both cases, the correctly reduced output value is computed.

Using Montgomery representation has the benefit of producing stricly normalized values in $[0; p-1]$ range, which allows fast subtractions and additions. Subtraction modulo p is implemented just like the last two lines of the reduction function, described above; it has an expected latency of 3 cycles, which is corroborated by the benchmarks.

Montgomery multiplication is a $64 \times 64 \rightarrow 128$ multiplication followed by Montgomery reduction. On recent Intel x86 CPUs, this multiplication uses the `mulx` opcode, which yields the low half of the result in 3 cycles (4 cycles for the upper half). Following operations involved in the Rust code above, we may expect that an optimal implementation will yield the result in a total of 10 cycles (assuming that all non-compute data movements are "free", as well as the zero-extension of a 32-bit value to 64 bits); again, this is what we achieve in the benchmarks.

For multiplications and squarings in $GF(p^5)$, it is beneficial to mutualize Montgomery reductions. For instance, the computation of the low coefficient of a product is performed as follows:

```
fn mul_to_k0(&self, rhs: &Self) -> GFp {
    let pp0 = (self.0[0].0 as u128) * (rhs.0[0].0 as u128);
    let pp1 = (self.0[1].0 as u128) * (rhs.0[4].0 as u128);
    let pp2 = (self.0[2].0 as u128) * (rhs.0[3].0 as u128);
    let pp3 = (self.0[3].0 as u128) * (rhs.0[2].0 as u128);
    let pp4 = (self.0[4].0 as u128) * (rhs.0[1].0 as u128);
    let zhi = (pp0 >> 64) + 3 * ((pp1 >> 64)
        + (pp2 >> 64) + (pp3 >> 64) + (pp4 >> 64));
    let zlo = ((pp0 as u64) as u128) + 3 * (
        ((pp1 as u64) as u128)
        + ((pp2 as u64) as u128)
        + ((pp3 as u64) as u128)
        + ((pp4 as u64) as u128));
    GFp(GFp::montyred(zlo + (zhi << 32) - zhi))
}
```

We recognize the five products (for a_0b_0, a_1b_4,...), each yielding a 128-bit output. The full linear combination could be up to 132 bits in length, which exceeds what can be stored in a `u128` variable. To keep to sizes for which Rust has portable types, and to avoid some contention on carry flags, we do the linear combination twice, on the low and high halves separately; the final expression which assembles `zlo` and `zhi` is a partial reduction (using the fact that $2^{64} = 2^{32} - 1 \bmod p$) which outputs a value that fits on 100 bits, well in range of the Montgomery reduction function.

For inversions, Legendre symbols and square roots, methods described in section 3.2 still apply, but since divisions in $GF(p)$ are much more expensive than multiplications (in out-of-VM architectures), inversions in $GF(p^5)$ are not as fast as inside the VM. We obtain an inversion in about 11.4 times the cost of a multiplication in $GF(p^5)$, which is very fast compared with the situation in prime fields, but still not fast enough to contemplate use of affine coordinates on the short Weierstraß curve.

5.2 Curve Operations

Using fractional (x, u) coordinates, we obtain generic point addition in 10M, and generic point doubling in 4M+5S; however, some optimizations can be applied to sequences of doublings, making it worthwhile to organize operations such that doublings happen in such long sequences.

We use window optimizations, similar to in-VM operations (see section 3.3). Inversions in $GF(p^5)$ are fast enough to allow normalization of window points to affine coordinates; mixed addition (addition between a point in fractional (x, u) and a point in affine (x, u) coordinates) has cost 8M instead of 10M. Conversion of t points from fractional to affine coordinates involves inverting $2t$ field elements, which can be done in a single inversion and $3(2t - 1)$ multiplications in $GF(p^5)$ (using Montgomery's trick of computing $1/u$ and $1/v$ as $(1/uv)v$ and $(1/uv)u$, respectively, and applying it recursively). With a 5-bit window, 64 point additions are needed, and using affine points saves 128 multiplications in $GF(p^5)$, while the normalization involves one inversion and 93 extra multiplications, making the operation worthwhile. Moreover, using affine coordinates for window points makes these points smaller in RAM, which speeds up constant-time window lookups. It is expected that the optimal window size will be 4 or 5 bits, depending on the target architecture; on the test x86 system, 5-bit windows seem slightly better.

For the special case of multiplying the conventional generator, multiple windows can be used (in our implementation, eight windows are used, for G, $2^{40}G$, $2^{80}G$,...) and they are precomputed, thereby avoiding the cost of conversion to affine. This operation is used when generating a new key pair, and when producing a Schnorr signature.

For signature verification, we apply the Antipa et al optimization[1] with Lagrange's lattice reduction algorithm[15]; the latter algorithm is implemented in about 20400 cycles on average. Verification is nominally on public values, and thus needs not be constant-time; we use direct array accesses for lookups. However, we do *not* use w-NAF representation of scalars, because a regular addition schedule favours long sequences of sequential doublings, for which performance is better than isolated doublings.

6 Conclusion

We presented an elliptic curve designed for a specific compute model. Although we use the Miden VM as a representative of that model, we expect this curve to be generally useful for other projects related to zero-knowledge proofs; curve design and implementation is also an interesting problem in its own right. As a general-purpose curve, ecGFp5 performance is not on par with the fastest standard curves (e.g. Curve25519), but is still decent enough: a single core on a laptop computer or a smartphone can generate or verify thousands of signatures per second.

Acknowledgements

We thank Bobbin Threadbare and Hamish Ivey-Law for providing the target context and useful discussions on optimized implementations in $GF(p^5)$, and Pierrick Gaudry for pointers and explanations on the fine details of curve attacks in extension fields.

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. S. Arita, K. Matsuo, K. Nagao and M. Shimura, *A Weil Descent Attack against Elliptic Curve Cryptosystems over Quartic Extension Fields*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E89-A, issue. 5, 2006.
4. E. Ben-Sasson, I. Bentov, Y. Horesh and M. Riabzev, *Scalable, transparent, and post-quantum secure computational integrity*, https://eprint.iacr.org/2018/046
5. C. Cremers and D. Jackson, *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman*, IEEE 32nd Computer Security Foundations Symposium (CSF), 2019.
6. C. Diem, *The GHS attack in odd characteristic*, Journal of the Ramanujan Mathematical Society, vol. 18, issue 1, pp. 1-32, 2003.
7. J.-C. Faugère, P. Gianni, D. Lazard and T. Mora, *Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering*, Journal of Symbolic Computation, vol. 16, issue 4, pp. 329-344, 1993.
8. P. Gaudry, *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem*, Journal of Symbolic Computation, vol. 44, issue 12, pp. 1690-1702, 2009.
9. P. Gaudry, F. Hess and N. Smart, *Constructive and destructive facets of Weil descent on elliptic curves*, Journal of Cryptology, vol. 15, issue 1, pp. 19-46, 2002.
10. M. Hamburg, *Decaf: Eliminating cofactors through point compression*, Advances in Cryptology - CRYPTO 2015, Lecture Notes in Computer Science, vol. 9215, pp. 705-723, 2015.
11. T. Itoh and S. Tsujii, *A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases*, Information and Computation, vol. 78, pp. 171-177, 1988.
12. A. Joux and V. Vitse, *Elliptic curve discrete logarithm problem over small degree extension fields. Application to the static Diffie-Hellman problem on $\mathbb{F}_q^5$*, Journal of Cryptology, vol. 26, issue 1, pp. 119-143, 2013.

13. *Polygon Miden*,
https://github.com/maticnetwork/miden
14. *Miden Assembly*,
version 0.2, accessed on 2022-02-21,
https://hackmd.io/YDbjUVHTRn64F4LPelC-NA
15. T. Pornin, *Optimized Lattice Basis Reduction In Dimension 2, and Fast Schnorr and EdDSA Signature Verification*,
https://eprint.iacr.org/2020/454
16. T. Pornin, *Double-Odd Elliptic Curves*,
https://eprint.iacr.org/2020/1558
17. J. Renes, C. Costello and L. Batina, *Complete addition formulas for prime order elliptic curves*, Advances in Cryptology – Eurocrypt 2016, Lecture Notes in Computer Science, vol. 9665, pp. 403-428, 2016.
18. E. Straus, *Addition chains of vectors (problem 5125)*, American Mathematical Monthly, vol. 70, pp. 806-808, 1964.