Montgomery Multiplication in Signed Redundant Representations

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

19 April, 2026

Abstract. In this paper, we explore the use of Montgomery multiplication with a multi-limb redundant representation of integers, in particular in combination with signed reduction factors. We develop techniques that are particularly suited to software platforms on which carry propagation is expensive, in particular RISC-V CPUs which lack hardware support for carries. We also show how to perform a whole-primitive range analysis that demonstrates that overflows are not possible, thus allowing liberal use of unreduced limb-wise additions and subtractions, which are small and fast. The implementation and analysis techniques are illustrated in a codegolfing exercise, to produce size-optimized implementations of ECDSA signature verification over NIST curve P-256; use of a virtual CPU with a custom instruction set with byte-size encoding ("bytecode") allows the production of an implementation as small as 848 bytes on x86 CPUs (in 64-bit mode); RISC-V (984 bytes), Armv8-A (1136 bytes) and portable C implementations (about 2200 to 2800 bytes) are also provided. In the process, an AI is utterly discomfited.

1 Introduction

Computations over modular integers are pervasive to many cryptographic schemes, including classic algorithms such as RSA[19] and Diffie-Hellman[8], elliptic curve signature schemes (ECDSA[13,1], EdDSA[5]) and key echange methods (ECDH[2]), pairing-based cryptography over pairing-friendly curves, but also lattice-based algorithms such as ML-KEM[14], ML-DSA[15], Falcon[17] (soon-to-be FN-DSA) and schemes relying on isogenies between supersingular curves. There is an ample literature on the subject; a most popular method for implementing arithmetic operations modulo some (odd) integer is the use of Montgomery multiplication[12].

This work is, at its core, an exploration of some variations on the use of *redundant* big integer representations with Montgomery multiplication: values are split into limbs which are strictly shorter than the machine word size, and they are allowed to extend beyond their nominal range. Redundant representations have long been studied, even before computers were invented[7], for efficient computations[6,9] and in particular for bounded carry propagation techniques in custom hardware (ASIC)[16]. Redundant representations with signed limbs can efficiently support operations modulo some special-format moduli[4]. The main potential advantage is the ability to perform additions and subtractions in a limb-wise way, without any reduction; moreover, the extra limb range can be leveraged to perform fewer carry propagations, which are detrimental to out-of-order and parallel execution, and also lack hardware support in some platforms (e.g. RISC-V). We furthermore consider use of *signed* values for

the Montgomery multiplicative factors, as well as signed or unsigned normalization. These variants are described in section 2.

To get a concrete context for the discussion, we implemented ECDSA signature verification over the NIST curve P-256 with optimization for code size rather than speed; this is a somewhat uncommon goal, which has some niche usages (e.g. boot-time verification of a signature on Flashable firmware, the verification code must fit in a small ROM). This was the occasion of a competition in which a developer (myself) was pitted against an LLM-based AI (Claude Code) over the course of about 8 days, with Intel x86 architecture as target (in 64-bit mode). The long history of the x86 platform made its instruction set especially rich in abstruse size optimizations, for which the pattern-matching behaviour of the LLM should be effective, especially since x86 assembly writing is a subject for which the Internet provides an extremely large corpus of examples. The bot was ultimately defeated, though it did provide some tough opposition; it also got it wrong at some point and produced code which deviated from the ECDSA specification. This is described in section 3. The outcome is instructive about what can and cannot be expected from LLMs, and how they may fail in the context of implementing cryptographic primitives.

In section 4, my own implementations are described, not only the x86-optimized code, but also 64-bit Arm (Armv8-A architecture) and 64-bit RISC-V (RV64 with I, M and C extensions), as well as C code with 64-bit and 32-bit words. The assembly implementations use a custom dedicated virtual machine with a few instructions that fit in a single byte each ("bytecode"), allowing to pack all the needed operations for ECDSA in a compact format.

Use of limb-wise additions and subtractions with no reduction implies that if too many such operations are performed in sequence, with no interleaved multiplications, then limb values may overflow. An implementation using that method can be deemed correct only if it can be proven never to reach an overflow condition. This requires an extensive and tight enough range analysis. In section 5, the bytecode from the previous section is leveraged into a Python-based interpreter which makes a comprehensive range analysis for all values throughout the entire ECDSA operation. This framework can moreover be reused over other word sizes, radices, and number of limbs; this is how my implementations with 32-bit words were produced.

All the implementations described here, as well as the range analysis framework, are available at:

https://github.com/pornin/small-ecdsa

2 Montgomery Multiplication and Variants

We describe here the classic Montgomery multiplication, and how it can be used in a redundant representation with signed limbs. For a well-known description of the classic algorithms, see the famous Handbook of Applied Cryptography ([11], section 14.3). Of all the discussion presented here, about none is really novel, except possibly the use of signed Montgomery factors f_i, and the pre-shifting of operands to ease the low/high split in long multiplication results.

2.1 Notations

For an integer $m > 0$, and two integers x and y, we write "$x \equiv y\,[m]$" to express that m divides $x - y$ without prescribing a particular range on either x or y. For modular *unsigned reduction*, we write "$x \bmod^+ m$" to designate the unique integer y such that $x \equiv y\,[m]$ and $y \in [0, m-1]$; for *signed reduction*, "$x \bmod^\pm m$" is the unique integer y such that $x \equiv y\,[m]$ and $-m/2 \leq y < +m/2$. Note in particular that when m is even, the lower end $-m/2$ is included in the signed reduction output, but the upper end $+m/2$ is not: this corresponds to how usual software platforms interpret signed words with two's complement notation, but it differs from the "$\bmod^\pm$" notation used in the ML-DSA scheme.

By extension, we may use these notations with *rationals*: for integers x, y and z such that y is coprime with m, we write $x/y \equiv z\,[m]$ to signify that $x \equiv yz\,[m]$; values $x/y \bmod^+ m$ and $x/y \bmod^\pm m$ denote integers z such that $x/y \equiv z\,[m]$, with $0 \leq z < m$ and $-m/2 \leq z < +m/2$, respectively.

2.2 Montgomery Multiplication and Representation

Let $m > 0$ be an integer; we want to perform arithmetic operations with integers modulo m. Let $R > m$ be some constant which is relatively prime to m, and such that Euclidean division by R on integers is easy. In all of the following, m is an odd integer, and R is a power of 2, so that the quotient and remainder of a division by R can be obtained with a right-shift and a truncation on the binary representation, respectively; however, the concept can be used with other values of R, as long as Euclidean division by R is efficient.

Montgomery reduction of an integer z consists in computing $f = -z/m \bmod^+ R$, then $z' = (z + fm)/R$. The computation of f (the "Montgomery factor") is done by multiplying z with the value $C = -1/m \bmod^+ R$, which depends only on the modulus m and is assumed to be precomputed (C exists because m is relatively prime to R); this multiplication is done modulo R, which is efficient (by definition of R). This specific factor f implies that $z + fm \equiv 0\,[R]$; the division $(z + fm)/R$ is therefore exact: the output z' is an integer.

If we consider values modulo m, then $z' \equiv z/R\,[m]$ (adding a multiple of m does not change the value modulo m). The point of Montgomery reduction is seen by considering ranges: since $0 \leq f < R$, we obtain that $|z'| < |z/R| + m$; in particular, if $0 \leq z < Rm$, then $0 \leq z' < 2m$.

Montgomery multiplication is the combination of plain integer multiplication and Montgomery reduction. Given two operands x and y, Montgomery multiplication computes $z = xy$ then applies Montgomery reduction and returns $z' = (xy + fm)/R$. If x and y are both in $[0, m-1]$, then $0 \leq z' < 2m$ and can be fully reduced modulo m with a single condi-

tional subtraction of m (subtract m from z' if and only if $z' \geq m$). We denote with "$*$" the Montgomery multiplication; for inputs x and y, we thus obtain that $x * y \equiv xy/R \ [m]$.

Montgomery representation of an integer x is another integer $\mathcal{M}(x)$ such that $\mathcal{M}(x) \equiv xR \ [m]$. Montgomery representation is compatible with addition, and with multiplication, provided that we use Montgomery multiplication: for all integers x and y, $\mathcal{M}(x) + \mathcal{M}(y) \equiv \mathcal{M}(x + y) \ [m]$, and $\mathcal{M}(x) * \mathcal{M}(y) \equiv \mathcal{M}(xy) \ [m]$. This finally yields the classic implementation method based on Montgomery representation:

- All values are kept in Montgomery representation, normalized to $[0, m-1]$. Conversion to Montgomery representation from a freshly decoded x integer is done by computing $x * (R^2 \bmod^+ m) = \mathcal{M}(x)$, the constant $R^2 \bmod^+ m$ being precomputed.
- Additions are done with a plain integer addition, followed by a conditional subtraction of the modulus m to ensure a result in $[0, m-1]$. Similarly, subtractions are done over plain integers, then adding back m if the result is negative.
- Multiplications are performed with Montgomery multiplication, which yields the correct result since we work with Montgomery representations. A conditional subtraction of the modulus m is applied at the end of the Montgomery multiplication to ensure that the output is in $[0, m-1]$.
- On output, right before encoding results, values are converted back from Montgomery representation to normal representation by applying Montgomery reduction. This can be implemented with a Montgomery multiplication: $x \equiv \mathcal{M}(x) * 1 \ [m]$.

One may note that if $x \in [0, m-1]$ then Montgomery reduction of x is also in $[0, m-1]$: we have $0 \leq f \leq R-1$, hence $0 \leq x + fm \leq m - 1 + (R-1)m = Rm - 1$; therefore, the final division by R necessarily yields a value strictly lower than m. This means that the final Montgomery reduction does not need the conditional subtraction of the modulus to ensure a canonical output. Notwithstanding, such operations are usually not time-critical, and it is easier to simply use the Montgomery multiplication routine with the constant 1, even though that implies some useless operations (plain multiplication by 1 is a no-operation).

2.3 Multi-Limb Montgomery Multiplication

For some cryptographic schemes (typically lattice-based schemes), all involved integers fit in a machine word and the implementation can follow the formulas described above. However, some others need to work with larger integers, which must then be split over several words. Let w be the machine word size, in bits; the implementation platform can compute additions and subtractions over values by working, implicitly, modulo 2^w. Each w-bit word can be considered as either *signed* or *unsigned*:

- In signed representation, the word contains a value in $[-2^{w-1}, 2^{w-1} - 1]$ (with two's complement notation for negative values).
- In unsigned representation, the word contains a value in $[0, 2^w - 1]$.

A given pattern of w bits can be converted between the two representations by implicit reinterpreting, which usually has no reality at the assembly level: the CPU does not care whether you think of some values as signed or not, it changes nothing to the additions, subtractions,

and multiplications modulo 2^w. Mathematically, if we number bits from 0 to $w-1$ (in low-to-high order), then unsigned representation means that bit $w-1$ has weight 2^{w-1}, but weight -2^{w-1} if the value is signed.

"Long multiplications" are multiplications with an output that spans over two words (that is, multiplications which are done over plain integers, *not* modulo 2^w). For these operations, signedness matters; thus, CPUs usually offer two instructions, one for multiplication of signed values, and one for multiplication of unsigned values[1]. For input integers a and b, each being the signed or unsigned interpretation of a w-bit word, long multiplication returns two words, the "low word" and the "high word". The low word is to be interpreted as an unsigned value $c_L \in [0, 2^w - 1]$; the high word is interpreted as value c_H with unsigned representation if both a and b are unsigned, or signed representation otherwise. We then have $ab = c_L + 2^w c_H$.

These operations map to the abilities of usual software platforms, at least at the hardware level. In programming languages, signed and unsigned interpretations are usually distinguished with types, but explicit conversions (e.g. with Rust's "`x as u32`" syntax, or Go's "`uint32(x)`") can be used to move between the two interpretations at no cost at the hardware level.

In the following, we will use a Python-based pseudocode notation with arithmetic operations on words specified with dedicated functions; in the descriptions below, we use x_s and x_u to denoted the contents of the word `x` interpreted as signed and unsigned, respectively (i.e. $-2^{w-1} \leq x_\mathrm{s} < 2^{w-1} - 1$, $0 \leq x_\mathrm{u} < 2^w$, $x_\mathrm{s} = x_\mathrm{u} \bmod^{\pm} 2^w$ and $x_\mathrm{u} = x_\mathrm{s} \bmod^{+} 2^w$).

`add(a, b)`	Addition: returns $a_\mathrm{u} + b_\mathrm{u} \bmod^{+} 2^w$.
`add_carry(a, b, c_in)`	Addition with carry: this computes $d = a_\mathrm{u} + b_\mathrm{u} + 1$ if c_in is `True`, or $d = a_\mathrm{u} + b_\mathrm{u}$ if c_in is `False`. Two values are returned: $d \bmod^{+} 2^w$, and the output carry c_out which is `True` if $d \geq 2^w$, or `False` if $d < 2^w$.
`sub(a, b)`	Subtraction: returns $a_\mathrm{u} - b_\mathrm{u} \bmod^{+} 2^w$.
`sub_borrow(a, b, c_in)`	Subtraction with borrow: this computes $d = a_\mathrm{u} - b_\mathrm{u} - 1$ if c_in is `True`, or $d = a_\mathrm{u} - b_\mathrm{u}$ if c_in is `False`. Two values are returned: $d \bmod^{+} 2^w$, and the output borrow c_out which is `True` if $d < 0$, or `False` if $d \geq 0$.
`lsh(a, n)`	Left-shift: returns $2^n a_\mathrm{u} \bmod^{+} 2^w$. Shift count `n` must be a non-negative constant integer. At the binary level, bits are shifted to the left (toward more significant places), and zeros are injected in the vacated low positions.
`rsh_s(a, n)`	Right-shift (arithmetic): returns $\lfloor a_\mathrm{s}/2^n \rfloor \bmod^{+} 2^w$. Shift count `n` must be a non-negative constant integer. At the binary level, bits are shifted to the right (toward less significant places) and copies of the sign bit (leftmost bit) of the input are injected in the vacated high positions.
`rsh_u(a, n)`	Right-shift (logical): returns $\lfloor a_\mathrm{u}/2^n \rfloor \bmod^{+} 2^w$. Shift count `n` must be a non-negative constant integer. At the binary level, bits

[1] RISC-V CPUs also have a third kind, for mixed signed/unsigned multiplications.

	are shifted to the right (toward less significant places) and zeros are injected in the vacated high positions.
`mul(a, b)`	Multiplication: returns $a_\mathrm{u} b_\mathrm{u} \bmod^+ 2^w$.
`longmul_s(a, b)`	Long multiplication (signed): this computes $d = a_\mathrm{s} b_\mathrm{s}$, then returns that value as two words: the low word $d \bmod^+ 2^w$ and the high word $\lfloor d/2^w \rfloor \bmod^+ 2^w$. The inputs are interpreted as signed values for the purpose of computing the product d.
`longmul_u(a, b)`	Long multiplication (unsigned): this computes $d = a_\mathrm{u} b_\mathrm{u}$, then returns that value as two words: the low word $d \bmod^+ 2^w$ and the high word $\lfloor d/2^w \rfloor \bmod^+ 2^w$. The inputs are interpreted as unsigned values for the purpose of computing the product d.
`is_zero(x)`	Comparison against zero: returns `True` if $x_\mathrm{u} = 0$, or `False` otherwise.

These functions correspond to hardware instructions offered by common CPUs, with some variations, in particular how carries and borrows are handled:

- On x86 and Arm CPUs, there is a single carry flag, stored in the flags register, which all instructions must share. Recent x86 support extra instructions (`adcx` and `adox`) which allow maintaining two interleaved addition chains with separate carries, though there is no corresponding support for subtractions.[2]
- On Arm CPUs specifically, the CPU carry flag is reversed for borrows: the carry flag is set when there is no borrow, and is cleared when there is a borrow. This is an implementation detail which is not relevant to Montgomery multiplication, since we do not mix carry flags between additions and subtractions. In this paper, we treat carries and borrows as Boolean values `True` or `False`.
- PowerPC CPUs (and later POWER CPUs) offer eight separate carry flags. It should be noted that some PowerPC implementations (especially the "PPE" core found in the PlayStation 3 and Xbox 360 consoles) handle only the first flag natively, and resort to slow microcode for the other seven flags.
- As opposed to PowerPCs' great trove of carry flags, RISC-V CPUs do not have a carry flag at all. For these CPUs, `add_carry()` requires no fewer than *five* instructions in all generality (but only two instructions if either the input carry is known to be `False`, or the output carry is ignored). The inefficiency of carry propagation on RISC-V was the initial motivation for this work.
- SIMD instructions (SSE2/AVX2/AVX-512 on x86, NEON on Arm, etc) typically do not have carry flags. Implementation using these instructions must resort to solutions similar to those for RISC-V code.

With these notations, the pseudocode in listing 1 shows how a classic multi-limb modular integer implementation would compute additions, subtractions and Montgomery multiplications. Each value x is represented in radix 2^w as an array of k words $(x_i)_{0 \le i < k}$, which are

[2] It should be understood that on modern *large* CPUs, with out-of-order execution, the physical CPU contains many separate carry flags to which the programming-level unique flag is dynamically mapped. Having a single carry still implies some constraints on both the compiler or assembly developer, and on how much parallelism the CPU can infer from the code.

Listing 1: Multi-limb modular arithmetics (classic representation)

```
# k limbs (unsigned), radix 2**w
# m0i = -1/m mod 2**w, for modulus m (odd)

# Add multi-limb integers a and b, with output carry (True on overflow).
def bigint_add_carry(a, b):
    d = [ZERO]*k
    carry = False
    for i in range(0, k):
        (d[i], carry) = add_carry(a[i], b[i], carry)
    return (d, carry)

# Subtract multi-limb integer b from a, with output borrow (True on overflow).
def bigint_sub_borrow(a, b):
    d = [ZERO]*k
    borrow = False
    for i in range(0, k):
        (d[i], borrow) = sub_borrow(a[i], b[i], borrow)
    return (d, borrow)

# d <- d + g*x
# g is 1 word, x is k words, d is at least k+1 words and large enough
def bigint_add_mul(d, g, x):
    delayed = ZERO
    for i in range(0, k):
        (cL, cH) = longmul_u(g, x[i])
        (cL, carry) = add_carry(cL, delayed, False)
        (cH, _) = add_carry(cH, ZERO, carry)
        (d[i], carry) = add_carry(cL, d[i], False)
        (delayed, _) = add_carry(cH, ZERO, carry)
    (d[k], carry) = add_carry(d[k], delayed, False)
    if carry:
        (d[k + 1], _) = add_carry(d[k + 1], ZERO, carry)

# Modular addition.
def mod_add(a, b):
    (d, carry) = bigint_add_carry(a, b)
    (e, borrow) = bigint_sub_borrow(d, m)
    return e if carry == borrow else d

# Modular subtraction.
def mod_sub(a, b):
    (d, borrow) = bigint_sub_borrow(a, b)
    (e, _) = bigint_add_carry(d, m)
    return e if borrow else d

# Montgomery multiplication.
def mod_montymul(a, b):
    t = [ZERO]*(2*k + 1)
    for i in range(0, k):
        bigint_add_mul(t, a[i], b)
        f = mul(t[0], m0i)
        bigint_add_mul(t, f, m)   # this ensures that t[0] = 0
        t = t[1:]                 # word shift: this is division by 2**w
    d = t[0:k]
    (e, borrow) = bigint_sub_borrow(d, m)
    return d if (is_zero(t[k]) and borrow) else e
```

implicitly considered unsigned (in $[0, 2^w - 1]$). The words (x_i) encode x with the rule that $x = \sum_{i=0}^{k-1} x_i 2^{iw}$. Modular integers are kept in canonical unsigned format: inputs and outputs of modular addition, subtraction and Montgomery reduction are always in $[0, m - 1]$, with m being the modulus.

The internal functions `bigint_add_carry()` and `bigint_sub_carry()` compute multi-limb additions and subtractions on non-modular integers, and return the result modulo 2^{kw} as well as an output carry or borrow; they are used for modular addition (`mod_add()`) and subtraction (`mod_sub()`) to ensure that the output is canonical. The internal routine `bigint_add_mul()` is used for Montgomery multiplication; it adds gx to a mutable value d, with g being a single word, x an integer over k words, and d being "large enough", meaning that it has at least $k + 1$ words, possibly $k + 2$ (it can be shown that within the Montgomery multiplication as implemented here, d is always large enough). Montgomery multiplication itself is `mod_montymul()` and uses $R = 2^{kw}$. Implicitly used parameters are the number of limbs k, the modulus m, and the Montgomery coefficient $-1/m \mod^+ 2^w$ (denoted `m0i` in the code). Since the inputs are canonical, the core of the Montgomery multiplication always returns a value in $[0, 2m-1]$ and a single conditional subtraction of the modulus (lines 55-56 in the code) is sufficient to ensure a canonical output.

Not shown in listing 1 are the decoding function (split of the input into k words of w bits), conversion to Montgomery representation (done with `mod_montymul()` and the precomputed constant $R^2 \mod^+ m$), Montgomery reduction (`mod_montymul()` with 1 as second operand), and reassembly of the words into a suitable output format, e.g. a sequence of bytes.

Some variants are possible; in particular, the implementation might not insist on a strictly canonical output, and may instead work internally with a somewhat larger range, e.g. $[0, R - 1]$ (all sequences of k words being then possible). Code working with a fixed compile-time modulus, as is typical of optimized elliptic curve implementations, may choose to unroll loops and use statically assigned registers rather than arrays for values; the routines may also be subject to different degrees of inlining. *A contrario*, for a large modulus, and especially for a modulus whose contents and even size are known only at runtime, actual arrays and indexing are likely to be used, and the two `bigint_add_mul()` routines called in each iteration of the `mod_montymul()` loop may be interleaved so as to minimize such memory traffic.

The salient point of the classic implementation is its reliance on carry propagation, both at the hardware level (the `add_carry()` and `sub_borrow()` primitives) and through the propagation of the "`delayed`" word on `bigint_add_mul()`. Modular addition and subtraction both involve two chains of instructions with linear dependencies (along each chain, the carry/borrow output of an instruction is used as input for the next), which reduces parallelism opportunities, especially when using SIMD instructions or out-of-order execution; on platforms without support for carries (such as RISC-V), these operations are particularly expensive, both in code size and in execution speed. Additions in `bigint_add_mul()` also represent long dependency chains (each value of the `delayed` variable depends on the output of the previous iteration, which itself depends on the previous value of `delayed`). The conditional choices ("`if`" clauses in `mod_add()`, `mod_sub()` and `mod_montymul()`, on lines 38, 44 and 56, respectively) may moreover involve an extra value scan to support a constant-time implementation, in case the involved values are secret. Our improved implementation method with a redundant representation was designed in particular to avoid most of these carry propagation issues.

2.4 Redundant Representation

A *redundant representation* of a big integer uses a radix which is strictly smaller than the word size. Here, we will use radix 2^s for some integer $s < w$ (with w being the word size). Redundancy comes from the idea that individual limbs may have values beyond the nominal $[0, 2^s - 1]$ range, which implies that a given integer may have several possible representations. Redundant representations are a well-known area (see section 1 for references). In the version described here, we are investigating the effect of some variants that involve use of signed values in various places.

In all generality, the mathematical description is the following. For a given integer x, we use a representation over k limbs in radix 2^s:

$$x = \sum_{i=0}^{k-1} x_i 2^{si}$$

The limbs are, at this level, plain integers which are not limited to the $[0, 2^s - 1]$ range; they can be larger than $2^s - 1$, or negative. The representation of x as (x_i) is not unique. We can do additions and subtractions in a limb-wise way:

$$x + y = \sum_{i=0}^{k-1} (x_i + y_i) 2^{si}$$

In a redundant representation, we can move some bits from a limb to the next, using:

$$x_i + x_{i+1} 2^s = (x_i \bmod^+ 2^s) + (x_{i+1} + \lfloor x_i / 2^s \rfloor) 2^s$$

In plain words, we can split a limb into its low s bits (a value in $[0, 2^s - 1]$) and its upper bits, which can be added to the next limb with a right-shift by s bits. Since limb values are potentially negative, we of course consider an arithmetic right shift here. A signed analog exists, in which we force the lower limb into $[-2^{s-1}, 2^{s-1} - 1]$:

$$x_i + x_{i+1} 2^s = (x_i \bmod^\pm 2^s) + \left(x_{i+1} + \frac{x_i - (x_i \bmod^\pm 2^s)}{2^s} \right) 2^s$$

This analog will be considered in an implementation variant ("signed normalization").

Montgomery multiplication uses the s-bit Montgomery coefficient $g = -1/m \bmod^+ 2^s$, which depends on the modulus m but not on the operands x and y. The process starts with an integer t_0, as a k-limb sequence $(t_{0,j})$, with all limbs equal to zero. Then, for $i = 0$ to $k - 1$, we define:

$$\begin{aligned}
u_{i,j} &= t_{i,j} + (x_i y_j \bmod^+ 2^s) + \lfloor x_i y_{j-1} / 2^s \rfloor \\
f_i &\equiv g u_{i,0} \ [2^s] \\
v_{i,j} &= u_{i,j} + (f_i m_j \bmod^+ 2^s) + \lfloor f_i m_{j-1} / 2^s \rfloor \\
t_{i+1,0} &= v_{i,1} + v_{i,0} / 2^s \\
t_{i+1,j} &= v_{i,j+1} \quad (\text{for } j \geq 1)
\end{aligned}$$

These equations are to be understood with j ranging from 0 to k (inclusive), with the convention that limbs with an out-of-bounds index (e.g. m_{-1} or x_k) have value zero. Intermediate values u_i and v_i expand over $k+1$ limbs. The values f_i are the Montgomery factors; note that they are defined above only as a congruence class modulo 2^s: the operation works regardless of which modular reduction is used. Classic Montgomery reduction would set $f_i = g u_{i,0} \bmod^+ 2^s$ (factor in $[0, 2^s - 1]$), but here we will use $f_i = g u_{i,0} \bmod^\pm 2^s$ (factor in $[-2^{s-1}, 2^{s-1} - 1]$). The core idea of the Montgomery multiplication still applies: if we work modulo 2^s, then:

$$\begin{aligned} v_{i,0} &\equiv u_{i,0} + f_i m \ [2^s] \\ &\equiv u_{i,0} + m(-1/m) u_{i,0} \ [2^s] \\ &\equiv 0 \ [2^s] \end{aligned}$$

so that the division of $v_{i,0}$ by 2^s (in the definition of $t_{i+1,0}$) is exact, and the dropped value $v_{i,0} \bmod^+ 2^s$ is in fact zero. This implies that $t_{i+1} = (t_i + x_i y + f_i m)/2^s$, and the k iterations collectively compute $t_k = (xy + fm)/2^{ks}$ for an integer f whose k-limb representation is (f_i). This follows the classic Montgomery multiplication for the values, with $R = 2^{ks}$, though most carry and upper bits propagations have been removed (only the upper bits of $v_{i,0}$ are propagated). Since the output limbs (in t_k) tend to be quite large, we follow Montgomery multiplication with a single upper bits propagation pass, which we will later call "normalization".

Listing 2 shows a pseudocode implementation of modular addition, subtraction and Montgomery multiplication in a redundant representation. We will now highlight some optimizations and other implementation variants.

Limb-wise additions and subtractions. The functions for addition (`mod_add()`) and subtraction (`mod_sub()`) operate in a limb-wise way, with no carry/borrow propagation; in fact, listing 2 uses no `add_carry` or `sub_borrow` at all, which improves speed and reduces code size on platforms that don't have hardware support for carries. The limb additions and subtractions are also independent of each other and can be carried in parallel, which may efficiently leverage SIMD units such as SSE2. The flip side of this parallelism is that limbs grow beyond the nominal range; if too many such operations are performed in a row, limbs may overflow the representable range in w-bit words. Most of the range analysis in section 5 is about proving that such overflows cannot happen in the code that uses these redundant integers.

Apart from limb growth, unreduced additions and subtractions also entail value growth: if the two source operands live in ranges $[a_L, a_H]$ and $[b_L, b_H]$, then their limb-wise sum is in $[a_L + b_L, a_H + b_H]$. Excessive value growth is *also* a concern, and calls for intermediate reduction steps that help prevent it. As will be detailed below, limb growth can be controlled with normalization, and Montgomery multiplication (which includes a normalization step) reduces both limb and overall value ranges.

Signed limbs. Limb values are interpreted as signed integers; this is apparent in the use of `longmul_s` in the Montgomery multiplications (line 21), and an arithmetic right-shift in normalization (line 33). Support of negative limb values allows a simple limb-wise subtraction; if limbs had to be kept non-negative at all times, then subtraction would involve either

Listing 2: Multi-limb Montgomery arithmetics (redundant representation)

```
# k limbs (signed), radix 2**s, with s < w
# sh1 = floor((w - s)/2)
# sh2 = w - s - sh1
# m0i_sh = (-1/m mod 2**s)*2**(w-s), for modulus m (odd)
# Two possible variants, depending on whether multiplication output limbs
# are normalized to [0, 2**s - 1] or to [-2**(s-1), 2**(s-1) - 1].

# Limb-wise addition.
def mod_add(a, b):
    return [add(a[i], b[i]) for i in range(0, k)]

# Limb-wise subtraction.
def mod_sub(a, b):
    return [sub(a[i], b[i]) for i in range(0, k)]

# d <- d + g*x  (limb-wise)
# g is 1 word, x is k words, d is at least k+1 words
def rr_add_mul(d, g, x):
    delayed = ZERO
    for i in range(0, k):
        (cL, cH) = longmul_s(g, x[i])
        d[i] = add(rsh_u(cL, w - s), add(d[i], delayed))
        delayed = cH
    d[k] = add(d[k], delayed)

# d <- a
# Limbs 0 to k-2 are normalized to [-2**(s-1), 2**(s-1) - 1] (if signed = True)
# or [0, 2**s - 1] (if signed = False).
def normalize(a, signed):
    d = [ZERO]*k
    cc = ZERO
    for i in range(0, k - 1):
        cc = add(a[i], rsh_s(cc, s))   # VARIANT: add old cc to d[i], not a[i]
        z = lsh(cc, w - s)
        if signed:
            z = rsh_s(z, w - s)
            cc = sub(cc, z)
        else:
            z = rsh_u(z, w - s)
        d[i] = z
    d[k - 1] = add(a[k - 1], rsh_s(cc, s))
    return d

# Montgomery multiplication.
def mod_montymul(a, b):
    b_sh = [lsh(b[i], sh2) for i in range(0, k)]
    t = [ZERO]*(2*k)
    for i in range(0, k):
        rr_add_mul(t, lsh(a[i], sh1), b_sh)
        f = mul(t[0], m0i_sh)
        rr_add_mul(t, f, m)
        t[1] = add(t[1], rsh_s(t[0], s))
        t = t[1:]
    return normalize(t, False)         # VARIANT: normalize(t, True)
```

carry propagation, or at least adding extra multiples of the modulus, which would entail further limb growth, as well as more instructions to execute.

Normalization. To contain limb growth, normalization and Montgomery multiplication are used. We say that a value $x = \sum_{i=0}^{k-1} x_i 2^{si}$ is *signed-normalized* (respectively *unsigned-normalized*) if limbs x_0 to x_{k-2}, i.e. all limbs except the most significant one x_{k-1}, are in the signed s-bit range $[-2^{s-1}, 2^{s-1} - 1]$ (respectively, the unsigned s-bit range $[0, 2^s - 1]$). The `normalize()` function transforms an input integer into another representation of the same integer value, but normalized. The shown pseudocode supports both signed and unsigned normalization, depending on the value of the `signed` parameter, but any actual implementation might not necessarily need both. The size-optimized implementations of ECDSA signature verification shown in sections 4 use only unsigned normalization; a variant described in section 5.3 uses both.

Normalization does not modify the integer value; it efficiently contains limb growth except for the top limb x_{k-1}, which can be said to receive the excess bits. Normalization does nothing about overall value growth, for which the main tool is Montgomery multiplication, which also reduces the range of the output, as explained in section 2.2. Section 5 includes more details on limb and value growth.

Normalization, as presented in `normalize()`, uses a sequence of successive operations with dependencies (the value `cc` is obtained from the previous addition and needed as input for the next). These dependencies could be removed by delaying the addition of the upper bits from the previous limbs (`cc`) to after the truncation of `z` to s bits, rather than before. In such a case, the normalization is only partial (output limbs may slightly exceed their nominal range) but partial normalization could be enough to contain limb growth, and the removal of dependencies can only help CPUs with out-of-order execution and SIMD implementations. Full normalization will still be needed for canonicalization purposes, typically right before encoding output values.

Pre-shifting. The `longmul` operations return the product over a pair of machine words; that value must then be split into two limb-sized parts. When the representation is non-redundant and the radix matches the word size, the two words returned by `longmul_u` are these low and high parts, and the split is implicit and free. This is no longer the case with a redundant representation. Given a product c which is obtained as two w-bit words c_0 and c_1, we have $c = c_0 + 2^w c_1$, and we want the low and high radix-2^s halves c_L and c_H such that $c_L \in [0, 2^s - 1]$ and $c = c_L + 2^s c_H$. The low half c_L can be obtained from c_0 by masking out the top $w - s$ bits, which can be done with a bitwise AND operation (which is inexpensive); however, c_H requires combining bits from both c_0 and c_1. Some CPUs have dedicated instructions to extract a word value straddling over two registers (e.g. `shld` on x86, or `extr` on Armv8-A), but some others do not, and computing c_H then needs two shifts and a bitwise OR (RISC-V, as usual, is in that situation). In order to make this split easier and faster, we use *pre-shifting*:

– Operands for a multiplication are left-shifted by some bits prior to the multiplication: the limbs of the first operand (`a[i]`) are left-shifted by `sh1` bits (line 49) and those of the second operand (`b[i]`) are left-shifted by `sh2` bits (line 46), for two fixed constants

`sh1` and `sh2` whose sum is equal to $w - s$. The constants must be chosen so that these shifts do not overflow the represented limb values, which depends on limb growth.
- The net effect of pre-shifting is that the `longmul_s` operation returns not the formal product c, but $2^{w-s}c$. In particular, this implies that the high output word c_1 now contains exactly c_H, so that no complicated word extraction is necessary. As for the low word c_L, it can be obtained with a single right-shift of c_0 (the `rsh_u` operation on line 22).
- For the addition of the modulus m multiplied by the Montgomery factor, all the pre-shifting is done in the factor, while the modulus is unshifted.

In total, pre-shifting requires $2k$ extra left shifts for the operands, but each of the $2k^2$ splits of `longmul_s` outputs only needs a single right-shift operation.

Signed Montgomery factor. A not immediately obvious feature of the Montgomery multiplication shown in listing 2 is that the Montgomery factor `f` is signed. Formally, the overall Montgomery factor f (in the computation of $(ab+fm)/R$) is obtained limb by limb, and each limb f_i is, in the classic description, a non-negative value (it is in $[0, 2^s - 1]$ when limbs are in radix 2^s). In the redundant representation:

- The factor f_i (in variable `f` in the code, line 51) is obtained by multiplying the low limb `t[0]` with the constant `m0i_sh` which is equal to $2^{w-s}(-1/m \bmod^+ 2^s)$ (that is, the Montgomery coefficient g, left-shifted by $w-s$ bits). A wrapping multiplication (modulo 2^w) is used. Thus, the output `f` really contains an s-bit value, but stored in a w-bit word and left-shifted by $w - s$ bits.
- The addition of the modulus multiple is done with the sub-function `rr_add_mul()` (the same function as for adding the partial product $a_i b$), which uses `longmul_s` (line 21). This interprets the w-bit word `f` as a signed value, in the $[-2^{w-1}, 2^{w-1} - 1]$ range. This value is equal to the formal f_i multiplied by 2^{w-s}, therefore f_i is really in the signed $[-2^{s-1}, 2^{s-1} - 1]$ range.

The fact that the f_i values are signed is crucial for the range analysis. It means that the overall f factor cannot be (much) larger than $R/2$ in absolute value, which makes Montgomery multiplication output values smaller, and helps with value growth. For instance, if m is a 256-bit integer, $w = 64$, $k = 5$, and $s = 54$ (values are over five limbs in radix 2^{54}), then $R = 2^{ks} = 2^{270}$; if the input operands a and b are at most $30m$ in absolute value, then $(ab+fm)/R$ is less than about $0.55m$ (in absolute value), which allows for up to 54 successive limb-wise additions before value growth reaches $30m$ again. With unsigned f_i values, Montgomery multiplication outputs could reach up to about $1.055m$ (with inputs at most $30m$) and only 28 unreduced additions would be allowed until a $30m$ value growth is again obtained.

Post-multiplication normalization. Montgomery multiplication, as presented in listing 2, uses the extra room in each limb to accumulate partial results without carry propagation. However, the resulting values can be quite large, and a normalization step is normally required after the core Montgomery loop. In `mod_monymul()`, the `normalize()` function is called with a `False` second parameter, to perform unsigned normalization. A variant would be to use signed normalization. Signed normalization yields smaller limbs (in absolute value), and thus allows more downstream unreduced additions and subtractions to be performed (with regard to limb growth and potential overflows), until the next multiplication.

However, signed normalization is slightly more expensive (there is an extra subtraction per limb, line 37), and unsigned normalization is usually also needed (for integer canonicalization purposes), so that size-optimized implementations will usually prefer to use only unsigned normalization throughout.

3 Human vs AI

In mid-March 2026, over the course of about 8 days, a human developer (myself) battled against an AI (Claude Code) in a contest for producing the most size-optimized implementation of ECDSA signature verification. On my part, one of the main points of the exercise was to find out whether the redundant representation with signed Montgomery factors was novel or not, the assumption being that the AI has extensive knowledge of everything that has ever been published on the Internet. Claude access and operation were provided by Steve Weis, who thought that the experiment was fun and had some spare compute tokens.

3.1 ECDSA

We consider ECDSA signature verification only, not key pair nor signature generation. In particular, all values are public, and there is no applicable notion of a side-channel information leak. A useful mental model for a size-optimized ECDSA signature verification code is a system that needs to verify a signature on its own reflashable firmware at boot-time; the verification code must then fit in a dedicated ROM block, hidden inside the CPU, where room is scarce[3].

We use NIST curve P-256. The point coordinates are expressed in the field of integers modulo $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, a 256-bit prime. The curve order is a prime integer n which is close to p (but slightly lower); we call *scalars* the elements of the field of integers modulo n. Each curve point (except the special point-at-infinity) has two coordinates (x, y), which are integers modulo p; they fulfill the curve equation $y^2 = x^3 - 3x + B$ for a given curve constant B. A conventional, fixed generator point G is given. The private key is a non-zero scalar d, and the corresponding public key is the point $Q = dG$. A public key point is commonly encoded with the "uncompressed" format over 65 bytes: one byte of value `0x04`, followed by the x and the y coordinates, over exactly 32 bytes each: the encoding of a coordinate is the unsigned big-endian encoding of the unique integer in $[0, p-1]$ corresponding to the coordinate value modulo p.

A signature is a pair of scalars (r, s). In this paper, we use the so-called IEEE p1363 format, in which a signature is expressed over exactly 64 bytes: 32 bytes for each of r and s, in that order. Each scalar is encoded as the unsigned big-endian representation of the unique integer in $[0, n-1]$ which matches the scalar value.

The signed message m is first hashed with some hash function; only the hash output is used for signature generation or verification. For curve P-256, the hash output must have size at least 256 bits (32 bytes), and only the first 256 bits are used; the SHA-256 hash function is commonly used for ECDSA with P-256. Signature verification is described in FIPS 186-5 ([13], section 6.4.2), and entails the following:

1. Decode the public point Q, verifying that:
   - the encoded point size is exactly 65 bytes;
   - the first byte has value `0x04`;

[3] In that specific case, there are in fact some alternatives, e.g. the ROM could contain a hash function implementation, but the signature verification code could be loaded dynamically from Flash; it would suffice to verify the hash of that code against a hash value hardcoded into the ROM space.

   - each provided x and y coordinate is the unsigned big-endian encoding of an integer in the $[0, p-1]$ range;
   - the two coordinates (x, y) fulfill the curve equation $y^2 = x^3 - 3x + B$.
2. Check that the provided hash value has size at least 32 bytes, and interpret the first 32 bytes as an integer $e \in [0, 2^{256} - 1]$ (using the unsigned big-endian convention). In the defined contest API, we will also require the verifier to enforce a *maximum* hash value size of 64 bytes (if it is larger than that, then it is probably not a hash value, which means that the calling application is using the code wrongly and unsafely).
3. Check that the signature size is exactly 64 bytes, the two halves of which encoding integers r and s which are in the $[1, n-1]$ range (take care that this involves checking that the integers are lower than n, but also that they are both non-zero).
4. Compute $u = e/s \bmod^+ n$ and $v = r/s \bmod^+ n$ (s is invertible modulo n because $0 < s < n$ and n is prime).
5. Compute the point $W = uG + vQ$. If that point turns out to be the point-at-infinity, then the verification stops and returns a failure.
6. Get the x coordinate of point W as a canonical integer x_W in the $[0, p-1]$ range.
7. The signature is valid if and only if $x_W \equiv r \ [n]$.

A tricky point is that x_W is obtained as an integer modulo p, but is then reinterpreted as an integer modulo n. This is why it is important to take care that it has been properly canonicalized in the $[0, p-1]$ range. Moreover, $n < p$, which means that interpretation of x_W modulo n entails an extra modular reduction (but it can be performed with a single conditional subtraction of n, since n is very close to p). Similarly, the hashed message e is turned into an integer modulo n but it could be greater than n: the implementation must tolerate an out-of-range value, or first reduce e modulo n (again with a conditional subtraction).

3.2 The Event

The formal goal for the implementations is a function that verifies an ECDSA signature over curve P-256; inputs are the encoded public key (Q, over 65 bytes), the signature (64 bytes), and the hashed message (hash value, 32 to 64 bytes). I defined a simple one-function API in C, shown in listing 3; it includes comments which define precisely the expected semantics. Notably, the input parameter sizes are provided explicitly, and must be checked by the function; also, the hash value length must be verified to be at least 32 bytes, but also at most 64 bytes. This file was provided to the AI.

The implementation target was a 64-bit x86 system (which matches the virtual machine to which Claude was given access, to run test versions), assuming a "bare metal" model (no standard library available, so everything had the be done locally, even moving memory blocks). Compatibility with the standard 64-bit x86 System V ABI (used in Linux systems under the architecture name "x86_64", though "amd64" is also encountered) was to be respected (the AI turned out to have a bit of trouble with that rule, it kept trying to use extension opcodes such as `mulx` and had to be reined in explicitly).

ECDSA signature verification on curve P-256 is not a novel thing; there are plenty of implementations around, so that we could assume that Claude would be able to make running code, if only by reproducing an existing implementation. Moreover, size-optimization on that particular platform is known to have lots of "local tricks" by leveraging the complicated instruction encoding, due to the very long history of that specific instruction set (e.g.

Listing 3: API for ECDSA signature verification (over NIST curve P-256)

```c
#ifndef TV_ECDSA_P256_VERIFY_H__
#define TV_ECDSA_P256_VERIFY_H__

#include <stddef.h>
#include <stdint.h>

#define TV_ECDSA_P256_KEY_SIZE 65
#define TV_ECDSA_P256_SIG_SIZE 64

/*
 * Verify signature 'sig' (size 'sig_len' bytes) against public key
 * 'pub' (of size 'pub_len' bytes) over hash value 'hv' (of size
 * 'hv_len' bytes). The hash value MUST have been obtained from a
 * cryptographically secure hash function with an output of size at
 * least 32 bytes (usually, SHA-256 is used). The signature format is
 * "raw": the two components (r,s) of the signature are encoded as
 * unsigned integer in big-endian convention, over 32 bytes each, and
 * concatenated in that order. The public key uses the "uncompressed"
 * format, of size exactly 65 bytes (the first byte must have value 0x04).
 *
 * Returned value is 1 on success (signature is valid) or 0 on
 * error. An error is reported in any of these cases:
 * - signature size is not correct;
 * - public key size is not correct;
 * - hash value size is less than 32 bytes or is greater than 64 bytes;
 * - signature does not have a valid format (i.e. either r or s is zero
 * or is not lower, as an integer, than the curve order).
 * - public key does not have a valid format (one of the coordinates
 * is out of its nominal ranges, or the point is not on the curve);
 * - signature verification algorithm fails.
 *
 * Exact rules follow FIPS 186-5, section 6.4.2, with the hash value 'hv'
 * being 'H = Hash(M)' as computed in step 2. This implementation adds an
 * explicit check that the provided hash value length does not exceed
 * 64 bytes, because that would indicate that the caller is not provding
 * a hash value but the message itself, which is insecure (the use of a
 * cryptographic hash function is necessary for the overall security).
 * In any case, the verification algorithm starts by truncating the hash
 * value to the curve size, which is 32 bytes for curve P-256, and the
 * extra bytes are simply ignored. Hash values _shorter_ than 32 bytes
 * are also rejected because they mechanically cannot provide the "128-bit"
 * security level that is usually expected from the curve (and FIPS 186-5,
 * section 6.1.1, makes such rejection mandatory).
 */
int tv_ecdsa_p256_verify(const void *sig, size_t sig_len,
        const void *pub, size_t pub_len,
        const void *hv, size_t hv_len);

#endif
```

even if instructions can work with any of the registers, the "historical" registers yield smaller encodings); we thus expected a pattern-matching engine like an LLM-based AI to be quite good at that task. To ensure correction of the result, Steve provided the AI with a fresh copy of relevant test vectors from project Wycheproof[20], which were designed to exercise many edge cases.

When the competition started, I already had a C implementation using 54-bit limbs in a redundant Montgomery representation, which I had manually translated to assembly, for a code size of about 2080 bytes. The only non-trivial (but still quite classic) size optimization was the use of a custom 16-bit instruction format to represent over 86 bytes the 43 operations involved in the complete point addition formulas from Renes, Costello and Batina[18]: these are additions, subtractions and multiplications over a handful of local variables, which are readily mapped to three-operand instructions (4 bits for each of two sources and one destination, and 4 bits for the operation itself). The initial productions from Claude used classic Montgomery representation (with four 64-bit limbs) and started with non-complete formulas over Jacobian coordinates. The AI quickly found and adopted its own 16-bit custom format; after being told by Steve to better look at existing formulas, it remembered that the Renes-Costello-Batina formulas exist. More grinding at local optimizations led it to about 1450 bytes.

Meanwhile, my own local optimization efforts had reduced my code to about 1640 bytes, which was not good enough. However, it allowed me to gain a finer understanding of what operations were actually needed for ECDSA signature verification; I knew from the start that a *really* reduced implementation would have to use a comprehensive custom instruction set, running in what amounts to an emulated CPU, with very small instructions, preferably over one byte each (thereafter called "bytecode"). Designing that instruction set required first knowing what instructions would be useful for the task at hand. At that point, I knew enough to attempt it, so I did, and got my code down to 1154 bytes. Steve conveyed that result to Claude, with some appropriate orders and taunting to make it grind futher. The AI switched to a non-Montgomery representation with 32-bit limbs, leveraging the special format of the involved moduli: for curve P-256, when using 32-bit limbs, then both the coordinate field modulus and the scalar field modulus (i.e. the curve order) are such that the most significant limb is equal to $2^{32} - 1$, which allows for a specialized reduction routine.

Claude and myself worked concurrently, finding and applying local optimizations. I was first to get my code below the symbolic threshold of 1000 bytes. After a while, the AI got to two implementations at 890 and 908 bytes (the latter being noticeably faster) but was getting stuck; Steve tried to nudge it toward trying some other representations, in particular using five 54-bit limbs, but it got nowhere (which was indirectly a confirmation that the kind of redundant Montgomery representation that is described in this paper was not in Claude's training set for ECDSA). At that time, my own code was down to 929 bytes (I later got it down to 875 bytes, and an alternate version at 848 bytes, but it was in the post-competition cleanup). Since the optimization efforts were slowing down, I decided to end it with a last parting shot: a stupidly slow alternative, which does not even use Montgomery representation at all, but allows to reduce the code size to 784 bytes (which I quickly shrunk to 766, then 732 bytes). The robot was stumped, and Steve conceded defeat in its name.

3.3 Why Did Claude Lose?

The AI can only use pre-existing knowledge (what is on the Internet, basically) and cannot exploit any new trick that has not been published beforehand, and is not easily inferrable through the kind of pattern matching that is at the core of how an LLM works. It is understandable, for instance, that it could not leverage the notion of a redundant representation with signed Montgomery factors. However, my final (and slow) code uses none of these mathematics; it is in fact very basic, and leverages only well-known methods that should have been accessible to Claude.

The fact that Claude did not find it is probably an artifact of how it works: mostly by "grinding" incremental optimizations. It quite early chose a 16-bit instruction format for translating curve point addition formulas (this is natural, everybody does it — I did too, in my first implementations), and then tried to augment it with extra instructions, but remained stuck on the 16-bit size. By constrast, my final code uses 8-bit instructions, which allow for expressing more behaviour in fewer bytes. Moving from 16-bit to 8-bit instructions is a structural jump which at first would make the code quite larger, even though it ultimately allows for a more compact solution, after sufficient optimization rounds. This is a kind of "stepping back" meta-optimization that humans can do but is apparently harder for AIs; the way it works for me is that when I am thinking about a particular optimization problem, I am *also* doing some meta-analysis in the back of my mind, always trying to fit the task in a larger context and considering entirely different strategies. This background thinking can keep on going while I perform other mechanical tasks that do not require my entire focus[4]. It seems that the current AIs do not have such background thinking abilities.

The "stupid" code is available in a separate repository[5]; since it is very expensive to use (runtime cost is around 255 million cycles!), we will not discuss it further in this paper. In all of the following, we will focus on the "non-stupid" code, which in particular leverages the improved Montgomery multiplication with a redundant representation, as explained in section 2.4.

3.4 Benchmarks and Domination

Near the start of the battle, I had provided not only the API, but also a loose notion that the code should not be so slow that it was unusable, with a fuzzy limit at one billion clock cycles; I also gave a few figures measured on my own code. Steve conveyed these values to Claude, which took it as a hint that speed was very important, and it kept expressing its optimizations with sentences such as "Thomas is DOMINATED!" with uppercase letters, a grim reminder that, as a large language model, this AI was trained, alas, on the Internet[6].

Notwithstanding, it was getting some results which looked slightly too good. After the event, I tried Claude's code on my own system, and found cost figures roughly 1.5 times higher than what the AI was reporting. Looking at how it was measuring its own performance, it turned out that Claude relied on the "time-stamp counter" (`rdtsc`) which is provided in

[4] As it were, I found the main optimization of that implementation, i.e. removing multiplications entirely, while folding laundry.

[5] https://github.com/pornin/codegolf-ecdsa

[6] The "domination" terminology is from the academic field of multi-objective optimization and relates to Pareto frontiers; but the uppercase taunting is from less well-behaved areas.

x86 CPUs, without any regard for frequency scaling. In older times (say, about 20 years ago), the time-stamp counter was faithfully counting actual CPU cycles, and `rdtsc` was a good way to measure cost of cryptographic primitives. However, CPUs have since deployed automatic frequency changes to adapt to load but also available voltage and current temperature; this goes both ways, the CPU being able to run at a *higher* frequency than its nominal speed, though not continuously, because that would induce overheating (Intel calls this feature "TurboBoost"). The time-stamp counter now runs at a fixed, nominal frequency matching the CPU baseline but not changing when the CPU actually runs either slower or faster. The bottom line is that for every clock cycle that Claude was measuring with `rdtsc`, the CPU had actually executed 1.5 cycles worth of instructions, making it underestimate its own costs by that factor 1.5. On my own test system, I use `rdpmc` to access performance counters, which return the true cycle count (but require the access to have been allowed by the system administrator); I also fully disabled TurboBoost[7].

So much for domination. The robot kept boasting about its superior speed, though it was, in hindsight, an illusion. It never really obtained better performance.

3.5 The Case of Test Vector 481

During post-event testing of Claude's output, a bigger and somewhat more concerning surprise was that when I tried it in my own test framework, it failed on one test vector. This was unexpected because the AI was working on the Wycheproof test vectors, which my test framework also uses, and the failed test was on one of these vectors. Was the AI really running its own tests? The explanation turned out to involve some historical quirks of ECDSA.

ECDSA, like its predecessor non-elliptic-curve DSA, represents signatures as a pair of scalars (r, s). The ECDSA standards (FIPS 186-5, ANSI X9:62) do *not* define a standard serialization of such pairs into bytes, so that several different, incompatible encodings have emerged. The simplest one is the IEEE p1363 format, which we use here in our ECDSA implementations; a widespread alternative is to store r and s into an ASN.1 SEQUENCE of two INTEGER values, which is then DER-encoded into bytes. The ASN.1-based format is the one used in, for instance, X.509 certificates. ASN.1 and p1363 are the two main conventions in widespread usage (there are a few others but they are not as widely used). Correspondingly, project Wycheproof contains two distinct lists of test vectors, for the p1363 format (file `ecdsa_secp256r1_sha256_p1363_test.json`) and for the ASN.1 format (file `ecdsa_secp256r1_sha256_test.json`).

Since we were using the p1363 format, Steve, quite logically, provided the AI with a copy of the test vectors in p1363 format. On my side, being somewhat of a perfectionist[8], I used *both*: some of the ASN.1 test vectors specifically exercise DER decoding issues, but all others use properly ASN.1-encoded signatures, and I could automatically convert these into the p1363 format, thus making a bunch of additional test vectors. One of these — test vector 481 — was part of a few test vectors added by David Benjamin in January 2026, designed specifically to detect a potential issue in an implementation trick, but David only provided

[7] I disabled TurboBoost primarily to keep the CPU cooler, which implies that the CPU fan is running slower and more silently; I hate fan noise. But a side-effect is that it makes my benchmarks much more stable and reliable.

[8] "Perfectionist" is a nice terminology to designate a raving maniac.

the ASN.1 version, not the p1363 version. Claude produced code that used this trick, and had the bug, but since it was working only with the Wycheproof p1363 test vectors, it did not notice it. My own test framework turned out to be more comprehensive (and I did not have that issue to begin with).

The bug is related to the final steps of the verification, when the x coordinate of W is extracted as an integer x_W, which is then reinterpreted as an integer modulo n. Internally, W uses some kind of fractional representation, typically projective coordinates, and reconstructing x_W requires computing a modular division (modulo p): $x_W \equiv X/Z\ [p]$ for two integers X and Z. Inversion of Z is somewhat expensive, both in computation time and in code size. To avoid these costs, the trick is to assert that an integer $r \in [0, n-1]$ may correspond to only two possible source integers $x_W \in [0, p-1]$ when they are reduced modulo n; these are r and $r+n$. We can thus backport the final two steps of ECDSA signature verification into integers modulo p, and implement them by checking whether $X/Z \equiv r\ [p]$ or $X/Z \equiv r+n\ [p]$, which can be further simplified into testing that $(X - Zr)(X - Z(r+n)) \equiv 0\ [p]$. Hurray, no more division!

This is a nice trick, with the slight drawback that it is, in fact, wrong, at least in the way it is presented above. A target integer $r \in [0, n-1]$ can be obtained from *up to two* possible integers in $[0, p-1]$, but usually only one is possible. If it so happens that $r < p - n$, then indeed $r + n < p$ and if x_W, as an integer, is equal to either r or $r + n$, then the signature will be valid. However, most r values are greater than $p - n$, and then $r + n \geq p$ and is no longer a valid target value for x_W. The optimization is applicable only if $r + n < p$, but some implementations omit that test and *always* accept $r+n$ as a valid target (modulo p), or accept it if $r + n$ is greater than p but "not too big" (e.g. it fits in 256 bits).

Test vector 481 exercises this potential issue: it uses a value r which is very slightly above $p - n$, and a verification point W whose affine x coordinate is $x_W = r + n - p$ (when canonicalized into $[0, p-1]$). Strict application of the ECDSA rules leads to the conclusion that the signature is invalid, but Claude's implementation accepts it as valid, which probably means that it is using the trick described above but lacks the required test that $r < p - n$.

Formally speaking, the AI's productions were never fully correct, and would have to be disqualified from the event. In a less rigid apprehension, we may consider that Claude leveraged an existing technique which was also wrong; the fact that a corresponding Wycheproof test vector exists at all shows that the bug is also encountered in the wild, and the AI lifted it, not knowing that it is incorrect (or missing that the range test on r was necessary).

3.6 Analysis of AI Usability

The "artificial intelligences" based on large language models, which are highly fashionable right now (however quaint it might seem to future readers), have a large number of objectionable externalities, such as the stupendous energy and water usage of large data centres; the disregard for any intellectual property or even courteous attribution for the sources that the AI reproduces; the ruthless and clumsy large-scale scraping of training data that is often indiscernable from downright denial-of-service attacks; etc. Additionally, programming is both a technique and an art, and exercises such as extreme code size optimization (colloquially known as "codegolf") are obviously more on the artistic than technical side; artistically, AI-generated code is some abhorrent slop, a repellent effusion of ugliness, in the words of Hayao Miyazaki it is "an insult to life itself". One can guess from this paragraph that I am not overly

fond of the whole AI endeavour. Yet I think that it is possible to examine the technical usability and utility of the tool in a specific task (here, implementing cryptographic primitives) dispassionately, independently of these considerations. We will thus try in this subsection to focus on the technical aspects of AI coding for cryptographic primitives.

On the plus side, Claude showed a strong ability at finding and applying local optimization tricks, wading with ease into the corners and nooks of the arcane encoding of the x86 instruction set. The ability to maintain a (mostly) functioning ECDSA verifier while renaming registers and moving functions around is impressive. We initially conjectured that the AI would be somewhat good at it, and it exceeded my expectations. It is also quite relentless in the "grinding" tasks.

On the minus side, it did make mistakes, as explained above: it unquestionably used an outdated benchmarking method, and it applied an incorrect optimization trick. A human developer might make the same mistakes, so this is not a sufficient reason to dismiss code-generating AIs entirely. A more interesting question is then: assuming that your robot did make some mistakes, what do you do? How should that be fixed? A salient point of the generated assembly code is that it is not exactly readable. The AI was instructed to add explanatory comments, and they can be used to somewhat infer how the computation happens in broad terms, but this is not sufficient to allow manual modifications, e.g. to fix the issue with test vector 481. Internally, Claude produced and modified its assembly code through custom code generators in Python, which adds an extra layer of unintelligibility. The only reasonable path to fixing the bug is to provide the test vector to the AI and ask it to fix its code. This raises three noteworthy points:

- If you want to modify such AI-optimized code then you need the AI again. This has become a dependency. You cannot readily remove the AI from your maintenance strategy, even if, for instance, the AI company decides to raise usage price by a factor of 10, or in the event that the AI company, being a startup, hits a financial wall and is wiped out into bankruptcy then liquidation. Human developers writing code which is not maintainable by other humans are a known issue; AIs are not immune to that, and cannot be assumed to always be around.
- AIs cannot learn. If a human developer makes a mistake, you can talk to that developer about it, and they will remember the mistake and how to avoid it; the same human might even proactively inform other humans about it, a process known as "teaching". This cannot be really done with AIs as currently deployed, because their "mind" is split into two parts: on the one side, during the training, the internal LLM was trained, at stupendous cost, on an extremely large set (at first order approximation, the entire Internet!) and can potentially "remember" all or most of it. On the other hand, the AI *instance* that you are using has about no memory at all. It can *read* all its training (which is ingrained into its emulated neural paths) but it cannot modify it. The only way for an LLM to remember anything is to put it in its local window, which is continuously fed back into the inference engine. This memory is entirely local to your specific instance; if you fire up another Claude session it will remember none of it. The window is also very small, and this is structural to the way it is processed (the window is repeatedly reinjected entirely as inference produces output tokens). The end result is that if you want an issue and its avoidance to become a somewhat permanent piece of knowledge, then you *must* turn it into a test vector that should be fed to all other AIs working in the same area.

– AIs can flood you with generated comments and documentation, but that does not necessarily lead to a better understanding of what the AI did. A human developer who found a nifty trick is likely to boast about it, make blog posts and videos and presentations at hacking conferences, and generally trying to demonstrate all the elegance of the method and how cunning and aesthetically pleasing it is[9]. The human will take great care to be understood by at least a few other people. You cannot expect that from an AI, which does not have any long-term planning and does not follow up on its personal communication goals — it does not have personal goals at all. This makes it hard to ensure preservation of knowledge. In other words, AIs cannot learn, but they are also bad at teaching anything which is not in their immutable training set.

Every technical area has its own specificities; for the quite specialized task of implementing cryptographic primitives, a well-known feature is that one cannot use functional tests for security. Unit tests are good at ensuring that a given piece of code behaves normally when processing normal data. However, cryptography postulates a sentient attacker who will purposefully try to exercise edge cases, especially some cases hiding in dark mathematical corners that are outside of reach of automated strategies such as fuzzing. The only known method to make a correct implementation is a combination of "being very careful" (the traditional way), and formally proving correctness in a way which is independently verifiable, most preferably with automated verification.

What the discussion above shows is that AIs are good at producing code which successfully passes a given set of unit tests. However, unit tests never offer a complete validation; there can always be some issues that will not be detected by the tests, even for very good and comprehensive sets such as Wycheproof. The fast exploration of numerous variants, which is the very thing that makes AI-generated code technically desirable, also makes them good at finding new kinds of bugs that bypass even comprehensive testing strategies. What this means is that when generating code in an adversarial context (which is the case for cryptography, by definition), an AI such as Claude should be used only if it can produce not only the code, but also a verifiable proof that the code is correct, e.g. using tools such as Lean[10].

Arguably, the same could be said of humans writing cryptographic implementations. The existence of subtly incorrect code is not a new problem; AIs only make it more apparent because they are faster at producing code, often operated without careful human review (AIs are being deployed *because* they are much faster, and careful reviewing is the first casualty in a search for speed), and not much amenable to personal improvements through local learning and teaching. Using an AI to generate a cryptographic implementation really augments the importance of having the AI also provide sufficiently detailed descriptions and proofs of correctness to enable independent validation.

Unfortunately, there is no proving method that is really complete (there is always some unproven interface between between the physical world and the world of abstract proofs); however, we can hope for proving correctness of some complicated parts in order to help a manual review by a human. Section 4 describes my own implementation in details, and in particular how it is structured around a custom bytecode which is leveraged in section 5 to perform a comprehensive and automated range analysis that demonstrates that the operations on big integers faithfully follow the mathematical description from section 2.4.

[9] This entire paper clearly shows how much guilty I am of this particular trait.

4 Size-optimized ECDSA Verification

4.1 Implementations and Performance

I made several implementations, for various architectures:

- A pure C implementation is provided; it is portable (no unaligned accesses, no reliance on any particular endianness) and comes in two flavours, which we designate under the names `gen64` (values over $k = 5$ limbs in radix 2^{54}) and `gen32` ($k = 12$ limbs in radix 2^{22}). `gen64` is available when the platform supports $64 \times 64 \rightarrow 128$ multiplications; `gen32` uses only $32 \times 32 \rightarrow 64$ multiplications.
- Optimized assembly implementations for 64-bit x86 (`amd64`), 64-bit Arm (`arm64`) and 64-bit RISC-V (`rv64`). All three use the same 5-limb format as `gen64`, with the exact same sequence of operations.
- An alternate implementation for 64-bit x86 (`amd64alt`) which uses twelve limbs and radix 2^{22}; it uses *almost* the same operations as `gen32`; the exact details are described in section 5.3.

For size and speed performance, we use three test platforms:

- 64-bit x86: Intel i5-8259U at 2.3 GHz (Coffee Lake core); TurboBoost is disabled.
- 64-bit Arm: Broadcom BCM2712 at 2.4 GHz (Arm Cortex A76 core); this is a Raspberry Pi 5 single-board computer (SBC).
- 64-bit RISC-V: StarFive JH7110 at 1.5 GHz (SiFive U74 core); this is a StarFive VisionFive2 SBC.

All three platforms run an up-to-date Ubuntu 24.04 operating system. Tests are done with C compilers Clang 18.1.3 and GCC 13.3.0, with compilation flags `-Os -fno-builtin -fno-asynchronous-unwind-table`; these flags imply optimization for code size rather than speed, no generation of calls to external functions such as `memcpy()`, and no generation of stack unwind tables (these tables improve debugging but otherwise have no effect on runtime, and they increase code footprint, so we remove them). The choice of C compiler and its options only impact the C implementations.

Table 1 reports the binary code size (in bytes, as reported by the `size` tool on the compiled object file) and the average runtime cost (in millions of clock cycles, measured with the relevant in-hardware performance counters).

We can make a few remarks on the performance figures reported in table 1:

- In general, GCC seems to be slightly better than Clang at optimizing for code size. This is however reversed for RISC-V, which would point at GCC being not too good, or making no special effort, at using the compressed 16-bit instructions (which is normally done by prioritizing some specific registers within the compiler's register allocation algorithm).
- RISC-V turns out to be good for low footprint code. This is probably a combination of the extreme RISC-ness of its instruction set, making it very friendly to compilers, combined with the 16-bit compressed instructions. By contrast, Armv8-A instructions are more powerful but use a strict 32-bit size.

Architecture	x86		Arm		RISC-V	
	size	cost	size	cost	size	cost
`gen64` (Clang)	3019	1.68	3054	3.40	2228	4.72
`gen64` (GCC)	2565	2.25	2764	3.38	2704	5.03
`gen32` (Clang)	2887	6.90	3168	6.99	2478	18.73
`gen32` (GCC)	2486	7.29	2888	7.31	2950	17.84
Assembly	875	4.59	1136	4.01	984	7.05
Assembly (alt)	848	13.64	-	-	-	-

Table 1: Binary size (bytes) and runtime cost (millions of clock cycles) for our optimized ECDSA signature verification code, on 64-bit architectures.

- The `gen32` code is normally a bit larger than the `gen64` code because it uses both signed and unsigned normalization. On x86, this is counterbalanced by the fact that some opcodes that process 32-bit values benefit from shorter encodings than their 64-bit counterparts, which is why `gen32` is somewhat smaller than `gen64` on x86. The same effect explains why `amd64alt` is 27 bytes smaller than `amd64`. Since no such effect was expected for Arm and RISC-V, no "alternate" assembly implementation was done for these platforms.
- In general, the Arm Cortex A76 is a powerful core (quad issue out-of-order execution with a deep pipeline), on par with the Intel Coffee Lake. This shows here as similar performance between the two platforms for the compiled `gen32` implementation. For `gen64`, though, the Arm CPU is substantially slower. This is due to a misfeature of that specific core with regard to 64-bit multiplications with a 128-bit output: these operations need two separate instructions which are *not* pipelined, and do not pair with each other. The net result is that this core can perform only one such long multiplication every 7 clock cycles. By contrast, the x86 CPU can fire one such instruction every cycle. The used RISC-V CPU is a weaker core (dual issue in-order execution) but it can execute one long multipication every 2 clock cycles, which is why it is not that much slower than the Arm for `gen64`.

The overall speed is of course poor, by usual standards, and even more so for the assembly code, which uses a custom virtual machine (described in the next subsections). Internal operations are expressed as loops which are not unrolled, so that most limb accesses are turned into array accesses, implying heavy memory traffic and extra index calculations; all of that is computationally expensive. Moreover, the used algorithms are not optimized for speed. We can make a rough estimate of the runtime cost by counting how many multiplications are needed in either the coordinate or the scalar fields:

- Field squarings use the same routine as field multiplication.
- All curve point doublings use the generic point addition routine (14 multiplications) instead of an optimized point doubling.
- The point addition routine is generic for inputs in projective coordinates, and does not leverage the fact that in some of the calls, the second operand is really in affine coordinates.

– Curve operations use a simple double-and-add algorithm with no lookup tables and no NAF or wNAF optimizations, so that each verification requires an average of 512 calls to the point addition routine.
– Three inversions are used, using Fermat's little theorem, with a basic square-and-multiply that makes more multiplications than would be needed (especially for the scalar field n: the exponent $n - 2$ has high Hamming weight).

In total, each verification uses about 8400 field multiplications, which is more than twice what could be expected from a speed-optimized implementation. However, an important counter-remark about speed for signature verification is that for a large input, most of the verification cost is spent hashing the data, not doing the elliptic curve computations, so that the cost values shown here would probably not matter much in a practical setup.

4.2 Virtual Machine

The size-optimized implementations in assembly are organized as a sequence of instructions for a custom virtual machine which just a few instructions, tailored at implementing the needed steps for that specific task. Instructions fit on one byte each (except for `_CALL`, which uses two bytes); the sequence of instructions is thus called here "bytecode". The bytecode is very compact and allows to pack the needed operations in a small size (217 bytes). To this we must add some constants (136 bytes for the encoded moduli p and n, the precomputed $2^{10}(-1/n \mod^+ 2^{54})$, and the conventional generator point coordinates), and the code which emulates the virtual CPU on the concrete hardware, as well as the data decoding routine to handle the initial input.

At the source code level, the bytecode is written with symbolic constants and macros, which abstract away the details of instruction encoding. In order to avoid conflicts between these macros and the assembler normal mnemonics (e.g. we do not want our "add" instruction to be confused with the CPU opcode `add`), all the symbolic constants and macro for the bytecode start with an underscore character. We reproduce this convention in the text below.

The virtual machine offers 32 variables, numbered from 0 to 31. Each variable can hold a big integer in redundant representation (k limbs in radix 2^s). Variable 0 is called the *accumulator* and is the destination, and one of the operands, of most computational operations. Other variables are accessed through symbolic constants (at the source code level) which abstract away the variable indices; some are initialized to some specific values in the VM startup code sequence:

`_M`	Current modulus (read-only, returns p or n in unsigned-normalized format; default modulus is n).
`_ONE`	Constant integer 1.
`_ZERO`	Constant integer 0.
`_Gx`	Generator point, x coordinate, Montgomery representation ($\mathcal{M}(G_x)$).
`_Gy`	Generator point, y coordinate, Montgomery representation ($\mathcal{M}(G_y)$).
`_Kr`	First signature half (r).
`_Ks`	Second signature half (s).
`_Ke`	Hash value, as an integer (e).

`_Qx` Public key point, x coordinate (Q_x).

`_Qy` Public key point, y coordinate (Q_y).

All these values are unsigned-normalized. The `_Kr`, `_Ks`, `_Ke`, `_Qx` and `_Qy` are freshly decoded from the input data, and have not been checked yet, i.e. they can be any integer in the $[0, 2^{256} - 1]$ range; it is up to the bytecode to verify that they are in their proper ranges and that for instance the public key point is a real curve point.

Other symbolic names are defined for most other variables, which are initialized to zero:

- `_Wx`, `_Wy` and `_Wz` are meant to be the projective coordinates of a work curve point W. Similarly, `_Hx`, `_Hy` and `_Hz` are the projective coordinate of a second curve point H. W and H will be used as the operands of the point addition routine.
- `_Bm` will receive $\mathcal{M}(B)$ (Montgomery representation of the curve constant B). It is not initialy set when the VM starts; the bytecode recomputes it from the generator coordinates (this saves space compared to having $\mathcal{M}(B)$ as a hardcoded constant).
- `_T0` to `_T9` are temporaries.

Values can be copied between the accumulator (value 0) and other variables with the following instructions:

`_LD x` Copy the contents of variable `x` into the accumulator.

`_ST x` Copy the contents of the accumulator into variable `x`.

Arithmetic operations use the accumulator as first (or unique) source operand, and as destination operand; some of them accept a second source operand, which is provided as an integer argument (the variable index, normally one of the defined symbolic constants):

`_ADD x` Add the contents of variable x to the accumulator (limb-wise addition).

`_SUB x` Subtract the contents of variable x from the accumulator (limb-wise subtraction).

`_MUL x` Multiply the accumulator with the contents of variable x. This is Montgomery multiplication; in other words, it assumes that the operands are in Montgomery representation. The currently set modulus is used.

`_MRED` Apply Montgomery reduction on the accumulator. This is a source code level alias for `_MUL _ONE` (Montgomery multiplication with the integer 1).

`_NORM` Unsigned normalization of the accumulator.

`_SMOD y` Set the modulus to either n and p, depending on the value of the parameter `y`. This value `y` is *not* a variable index, but a conventional designation of the relevant modulus. Correct values of `y` can vary depending on the target architecture; thus, this instruction is normally used through the aliases `_MODN` and `_MODP`, which set the modulus to n and p, respectively. The current modulus is used in `_MUL` and can be read from variable `_M`.

A single loop facility is provided. These loops do not nest: only a single loop may be active at any given time. Each loop performs exactly 256 iterations, and internally maintains a loop counter that goes from 255 down to 0.

`_FOR` Start the loop. The counter is reset to 255, and the loop iteration point is set to the instruction that immediately follows the `_FOR` instruction.

`_NEXT` If the counter is not 0, then it is decremented, and the instruction pointer is set to the current loop iteration point (just after the `_FOR` that started the current loop). If the counter is 0, this instruction does nothing. Note that there could be several `_NEXT` instructions for the same `_FOR`.

`_FNEXT` If the counter is exactly 255 then `_FNEXT` has the same effect as `_NEXT`. Otherwise, it does nothing. This instruction is used to skip the first iteration of a loop.

Conditional execution uses three "skip" instructions: for each, if the represented condition is true, then the instruction that immediately follows is skipped (execution "jumps over" that following instruction); otherwise, the instruction that follows is executed.

`_SKIPBITZ x` Skip the next instruction if bit c of the value in variable `x` is zero. c is the current loop counter value.

`_SKIPNZ` Skip the next instruction if the accumulator contents are non-zero.

`_SKIPNEG` Skip the next instruction if the accumulator contents are negative.

`_SKIPBITZ` is the only instruction that really uses the loop counter c; it is of course meant to implement square-and-multiply and double-and-add algorithms. Since this uses the binary representation of the limbs in memory (this really tests bit $c \bmod^+ s$ of limb $\lfloor c/s \rfloor$), this assumes that the test variable `x` is in unsigned-normalized format. Similarly, `_SKIPNZ` really tests whether all limbs are zero, and `_SKIPNEG` only looks at the sign of the most significant limb (limb $k-1$): the accumulator contents must be unsigned-normalized for these tests to mathematically make sense.

A subroutine facility is provided:

`_CALL label` Call the subroutine that starts at the provided assembly-level `label`.

`_RET` Return from the current subroutine.

In the instruction encoding, an unsigned 1-byte offset is used, so that the subroutine must be located *after* the call site in the address space (this prevents recursion). Called subroutines may themselves call other subroutines (no nesting limitation). There is no facility to pass arguments, the caller and the callee must conventionally agree on which of the VM variables are used for the subroutine parameters.

Finally, two instructions exit the virtual machine unconditionally, with either a success status (signature is valid) or a failure status (signature is not valid):

`_OK` Exit the VM with a success.

`_FAIL` Exit the VM with a failure.

4.3 ECDSA Bytecode

The VM instructions are enough to implement all needed operations for ECDSA signature verification. When the VM starts, the calling code has already performed initialization steps, which entail the following:

– The source signature was verified to have length exactly 64 bytes. The two halves r and s have been decoded (as plain 256-bit integers) into the `_Kr` and `_Ks` variables; the values are unsigned-normalized and non-negative, but at this point they may be zero, and they may be as large as $2^{256} - 1$.
– The public key was verified to have length exactly 65 bytes, and its first byte to have value `0x04`. The remaining 64 bytes have been decoded into `_Qx` and `_Qy` (values are unsigned-normalized and non-negative, but not otherwise checked).
– The hash value was verified to have length at least 32 bytes and at most 64 bytes, and the first 32 bytes have been decoded into `_Ke` (unsigned-normalized, non-negative).
– All other variables have been initialized as per the VM semantics (current modulus is n; variables `_ZERO`, `_ONE`, `_Gx` and `_Gy` are set to their respective values, all others are initialized at zero).

Listing 4 shows the `check_scalar` subroutine, whose role is to verify that the value currently in the accumulator is a proper input scalar value, i.e. an integer strictly in the $[1, n-1]$ range. This subroutine assumes that the current modulus is n and that the value in the accumulator is unsigned-normalized. As this is part of some assembly source code, we can use comments (introduced by `#`) and we can also use the semicolon (`;`) to separate instructions that are specified on the same source line. The first `_SKIPNZ` will skip the subsequent `_FAIL` if the value in the accumulator is non-zero: this is the check that the scalar is not equal to zero. The next three instructions verify that subtracting the current modulus yields a negative integer; note the use of unsigned-normalization to make sure that the top limb reliably indicates the sign of the overall value. If both `_FAIL` instructions were skipped, then `_RET` is reached and the subroutine returns to the caller.

Listing 4: Bytecode: verification that an input value is a proper non-zero scalar.

```
# BYTECODE: Check that a value (just decoded) is non-zero and less than
# the modulus. This is meant for scalars; it uses the current modulus.
# Input:
#    acc   value to check
# Output:
#    acc   x - m if source value is x and modulus is m
check_scalar:
        _SKIPNZ ; _FAIL ; _SUB _M ; _NORM ; _SKIPNEG ; _FAIL ; _RET
```

Listing 5 takes an input value (in the accumulator) which is assumed to be a Montgomery representation $\mathcal{M}(x)$ of some value x, and it returns x in canonical representation (value is in $[0, m-1]$ if the current modulus is m, and it is unsigned-normalized). This subroutine inherently leverages the fact that we use signed Montgomery factors: since the (f_i) are signed values in $[-2^{s-1}, 2^{s-1} - 1]$, the absolute value of f is such that:

$$|f| \leq 2^{s-1} \sum_{i=0}^{k-1} 2^{si}$$

where the limit is equal to γR for some constant $\gamma \approx 1/2$. The main result is that after Montgomery reduction, the output value may be negative, but in absolute value it will be lower than the modulus m, and it can be canonicalized into $[0, m-1]$ with a single conditional addition; furthermore, the relevant test in that case is whether the value is negative, which is readily performed with `_SKIPNEG`. In the range analysis (section 5) we will also prove that our canonicalization always works (i.e. always returns a value in the expected $[0, m-1]$ range).

Listing 5: Bytecode: Montgomery reduction with canonicalization.

```
# BYTECODE: Fully reduce a value and return the canonical representation.
# Input is expected in Montgomery representation.
# Input:
#    acc
# Output:
#    acc
# Clobbers: none
canonicalize:
        # Get out of Montgomery representation; this also returns a
        # normalized value whose absolute value is less than m.
        _MRED
        # If the value is non-negative then it's canonical.
        _SKIPNEG ; _RET
        # Add the modulus to get a canonical value.
        _ADD _M ; _NORM ; _RET
```

Listing 6 shows how modular inversion is computed: this is a simple use of Fermat's little theorem: the input (accumulator) is raised to the power $m-2$, for modulus m. The exponent is set in variable `_T1`, by subtracting 2 from the modulus; one optimization here is that for both used moduli p and n, this limb-wise subtraction will not make the value unnormalized (the lowest limb of both p and n happens to be larger than 2) so we do not need an extra `_NORM` on the modulus.

An important point here is that since all multiplications are Montgomery multiplications, this really performs inversion on Montgomery representations: given $\mathcal{M}(x)$, the routine returns $\mathcal{M}(1/x \bmod^{+} m)$. A basic square-and-multiply algorithm would enumerate all bits of the exponent, and start with a value set to $\mathcal{M}(1)$. We do not have the constant $\mathcal{M}(1)$ in the virtual machine (it was not part of the constants initially set). Instead, we start the square-and-multiply with the input $\mathcal{M}(x)$ itself, and skip the top bit (bit 255) of the exponent, under the assumption that this exponent bit has value 1 (which is the case for both moduli n and p). This is the reason why the `_FNEXT` instruction exists: it was defined precisely to support this skip of the first iteration in computing inversions. Supporting `_FNEXT` in the emulator requires less code space than embedding the constant $\mathcal{M}(1)$.

With `check_scalar`, `canonicalize` and `invert_mod` defined, we can look at the start of the main bytecode sequence, which is shown in listing 7. The current modulus is n (this was set during VM initialization) so all operations here work on scalars. This sequence first checks that the decoded r and s values are in $[1, n-1]$ by using `check_scalar` on both. Then s is inverted (modulo n) and that inverse is multiplied with e and r, yielding two

Listing 6: Bytecode: modular inversion.

```
# BYTECODE: Modular inversion. The current modulus is used.
# Input:
#    acc   value to invert
# Output:
#    acc   computed inverse (zero if input is zero)
# Clobbers: T0, T1
invert_mod:
        # Copy source value into T0.
        _ST _T0
        # Set T1 to exponent, which is m-2 (canonical) for modulus m.
        # Subtraction is limb-wise but for the two involved moduli (p and n)
        # this does not overflow any limb, so m-2 is still canonical.
        _LD _M ; _SUB _ONE ; _SUB _ONE ; _ST _T1
        # Set accumulator to source.
        _LD _T0
        # Start iterator and consume first iteration
        _FOR ; _FNEXT
        # Square current value (except on first iteration)
        _MUL 0
        # If current bit is 1, multiply accumulator by operand
        _SKIPBITZ _T1 ; _MUL _T0
        # Loop until finished, then exit
        _NEXT ; _RET
```

Listing 7: Bytecode: entry sequence.

```
bytecode_entry:
        # Check that r and s are both non-zero and below n
        _LD _Kr ; _CALL check_scalar
        _LD _Ks ; _CALL check_scalar
        # Invert s. Ks currently contains Mont(s/R) (Montgomery
        # representation of s/R); inversion returns Mont(R/s).
        # Accumulator currently contains s - n = s mod n, which is fine for us.
        _CALL invert_mod ; _ST _Ks
        # Compute u = e/s (canonicalized, into Ke).
        # Since Ke contains Mont(e/R) and we have computed Mont(R/s),
        # a Montgomery multiplication yields Mont((e/R)*(R/s)) = Mont(u),
        # which we can canonicalize.
        _MUL _Ke ; _CALL canonicalize ; _ST _Ke
        # Compute v = r/s (canonicalized, into Ks).
        _LD _Ks ; _MUL _Kr ; _CALL canonicalize ; _ST _Ks
```

values u and v which are canonicalized (and stored into `_Ke` and `_Ks`, respectively — `_Kr` is unmodified).

As explained previously, `invert_mod` really works on Montgomery representations, and there is no apparent conversion to Montgomery representation here. Indeed, such a conversion would be done with a `_MUL` with the constant $R^2 \mod^+ n$ (as explained in section 2.2), but that constant is not provided to the VM. It so happens that such a conversion is not needed here. An integer x is also the Montgomery representation of $x/R \mod^+ n$; thus, the inputs r, s and e, stored in `_Kr`, `_Ks` and `_Ke` respectively, are also the values $\mathcal{M}(r/R)$, $\mathcal{M}(s/R)$ and $\mathcal{M}(e/R)$. The call to `invert_mod` on $\mathcal{M}(s/R)$ computed the inverse over Montgomery representations, i.e. $\mathcal{M}(R/s)$. Then, the `_MUL` call with `_Ke` (which contains $\mathcal{M}(e/R)$) yielded $\mathcal{M}(R/s) * \mathcal{M}(e/R) = \mathcal{M}(e/s)$. The `canonicalize` routine uses Montgomery reduction (`_MRED`) to convert the input out of Montgomery representation, and thus yields $e/s \mod^+ n$ which is exactly the formal value u that is needed for the rest of ECDSA verification. Similarly, the `_MUL` instruction between r and the inverse of s computes $\mathcal{M}(r/R) * \mathcal{M}(R/s) = \mathcal{M}(r/s)$ and canonicalization produces the correct value v. At no point is explicit conversion to Montgomery representation needed; it is done "naturally" as a side-effect of inversion through Fermat's little theorem. We therefore do not need the constant $R^2 \mod^+ n$.

As will be shown below, a similar optimization applies to curve points, thanks to the use of projective coordinates. The constant $R^2 \mod^+ p$ can also be omitted. Not needing these two constants saves at least 64 bytes in the overall implementation size[10].

At this point, values u and v have been computed and are in canonical format, which is appropriate for testing individual bits in a double-and-add algorithm for curve point multiplications. The next part of the main bytecode is shown in listing 8: it switches the current modulus to p, then proceeds to compute the value $\mathcal{M}(B)$ (Montgomery representation of the curve constant B) by using the curve equation and the provided generator point coordinates. The generator coordinates were set in Montgomery representation during the VM initialization, but we will afterwards need them in non-Montgomery representation, so they are converted with Montgomery reduction (`_MRED`).

The public key point Q has not been verified yet: the two provided coordinates (Q_x, Q_y) must be checked to be in the proper range $[0, p-1]$, and to fulfill the curve equation $y^2 = x^3 - 3x + B$. This is done by the subroutine `check_point`, which is shown on listing 9. Note the use of `_MRED` (line 35) before comparing a value with zero: since `_SKIPNZ` actually checks that all limbs are zero, this requires ensuring that the only possible representation of zero (modulo p) is the all-zeros limb sequence. This is ensured by Montgomery reduction, as a side-effect of using signed Montgomery factors: the output of Montgomery reduction is normalized, and while it is not *fully* reduced (some values modulo p accept several representations), it still is in a range $[-\gamma p, +\gamma p]$ for some $\gamma < 1$, which implies that only one representation is possible for the modular zero. We could have used `_CALL canonicalize` here, but a `_CALL` is two bytes, while `_MRED` is only one.

With the help of `check_point`, we can proceed with the rest of the ECDSA verification code, as shown on listing 10. It starts with calling `check_point` to verify the public key

[10] Amusingly, some of Claude's comments on its own code listed the space savings of not including the R^2 constants as the main motivation for not using Montgomery multiplication; as it turns out, they are not needed anyway.

Listing 8: Bytecode: computations of curve constants.

```
        # We have u and v; we now switch to modulus p for curve computations.
        _MODP

        # Compute Bm = b*2^270 mod p from the generator coordinates. At
        # this point, (Gx,Gy) are still in Montgomery representation.
        #    b = Gy^2 - Gx^3 + 3*Gx
        _LD _Gx ; _MUL _Gx ; _MUL _Gx ; _ST _T0
        _LD _Gy ; _MUL _Gy ; _SUB _T0 ; _ADD _Gx ; _ADD _Gx ; _ADD _Gx
        _ST _Bm

        # Convert (Gx,Gy) out of Montgomery representation (so that
        # (Gx:Gy:1) is a valid triplet).
        _LD _Gx ; _MRED ; _ST _Gx
        _LD _Gy ; _MRED ; _ST _Gy
```

Listing 9: Bytecode: verification of the public key point.

```
# BYTECODE: Check decoded point. This function verifies that the two
# coordinates are in [0,p-1] and that they fulfill the curve equation.
# Input:
#    current modulus MUST be mod_P
#    Qx   candidate x coordinate (affine, non-Montgomery)
#    Qy   candidate y coordinate (affine, non-Montgomery)
# No output. A failure is triggered if the point is incorrect.
# Clobbers: T0, T1, T2
check_point:
        # Check that Qx is lower than p. It was just decoded, and thus
        # we can simply subtract the modulus and check for sign.
        _LD _Qx ; _SUB _M ; _NORM ; _SKIPNEG ; _FAIL
        # While it's loaded in the accumulator, compute X^3 and store
        # it into _T1.
        _MUL _Qx ; _MUL _Qx ; _ST _T1
        # Check that Qy is lower than p (same treatment as Qx).
        _LD _Qy ; _SUB _M ; _NORM ; _SKIPNEG ; _FAIL
        # Compute Y^2 and store it into T2.
        _MUL _Qy ; _ST _T2
        # We use (X:Y:Z) = (x:y:1) as coordinates, but the projective
        # coordinates are supposed to be Montgomery representations, so
        # we really set X = x/R, Y = y/R and Z = 1/R.
        # We must check the curve equation:
        #    Y^2*Z = X^3 - 3*X*Z^2 + b*Z^3
        # which we rewrite as:
        #    ((3*X - b*Z)*Z + Y^2)*Z - X^3 = 0
        # We have b (in Montgomery representation) in constant Bm; each
        # multiplication by Z is a Montgomery reduction.
        _LD _Bm ; _MRED                 # acc <- b*Z
        _SUB _Qx ; _SUB _Qx ; _SUB _Qx  # acc <- b*Z - 3*X
        _MRED                           # acc <- (b*Z - 3*X)*Z
        _SUB _T2                        # acc <- -(((3*X - b*Z)*Z) + Y^2)
        _MRED                           # acc <- -(((3*X - b*Z)*Z) + Y^2)*Z
        _ADD _T1                        # acc <- X^3 - (((3*X - b*Z)*Z) + Y^2)*Z
        _MRED ; _SKIPNZ ; _RET          # success if acc == 0
        _FAIL
```

point Q; then it computes the point $W = uG + vQ$ with a basic double-and-add (combined) algorithm:

1. W is initialized to the curve neutral element (the point-at-infinity).
2. For $i = 255$ down to 0:
   (a) $W \leftarrow W + W$ (point doubling)
   (b) If bit i of u is non-zero, then: $W \leftarrow W + G$
   (c) If bit i of v is non-zero, then: $W \leftarrow W + Q$

The `_FOR` loop and the `_SKIPBITZ` instruction are used here to perform these operations. All curve point additions and doubling use the `point_add_to_W` routine, which we'll detail later on. Once W has been computed, its x coordinate (x_W) is extracted, which requires a modular inversion, since projective coordinates are used: `_Wx` and `_Wz` contain, respectively, the Montgomery representations of values X and Z such that $x_W = X/Z$. A call to `invert_mod`, then a `_MUL`, and finally canonicalization yield x_W in the format required for correct ECDSA processing: non-Montgomery representation, and canonical in the $[0, p-1]$ range.

Once x_W is obtained as an integer, the rest of the code switches back to the modulus n, and subtracts r (which is still in `_Kr`). Testing this value against zero is done with a call to `_MRED`, which, as previously explained, ensures that only the all-zeros representation is possible for a modular zero, thus allowing `_SKIPNZ` to reliably test the value.

A non-trivial optimization, which is explained on lines 36-39, is that we do not need to handle specially the case of W being the point-at-infinity. As per the ECDSA standard, if W is the point-at-infinity, then the signature is invalid and must be rejected. This is what happens here: in that situation, $Z = 0$, and inversion (using FLT) returns zero, so that the recomputed x_W coordinate ends up being zero (modulo p, but also as a canonical integer in $[0, p-1]$). Since value r was previously verified to be non-zero, this implies that x_W cannot be equal to r modulo n, thus leading to a `_FAIL` instruction, which is exactly what we want here.

The final element of the bytecode is the `point_add_to_W` routine, which is used both for point doublings, and for adding either G (the curve generator) or Q (the public key). This routine is shown on listing 11. It uses projective coordinates $(X{:}Y{:}Z)$ to represent the affine point $(X/Z, Y/Z)$. The point-at-infinity uses $(0{:}Y{:}0)$ for any $Y \neq 0$; for all non-infinity points, $Z \neq 0$. The code is a straightforward translation of the Renes-Costello-Batina formulas ([18], algorithm 4) which are complete for curve P-256, since that curve has no point of order 2. The source formulas are expressed as three-operand operations (two sources, one destination), each being mechanically converted into three instructions (load first source operand into the accumulator, do the operation with the second source operand, store back the result into the destination operand), which would lead to a bytecode size of 130 bytes (including the final `_RET`). However, in many places the result of an operation is immediately used as source for another, which allows us to remove some redundant loads and stores; *in fine*, the `point_add_to_W` routine uses 85 bytes of code space, which is actually *shorter* than the 86 bytes needed for 16-bit instructions: even for this routine where 16-bit opcodes seem most adequate, the 8-bit opcodes are more size-optimized.

One final noteworthy point here is how the $R^2 \bmod^+ p$ constant for conversions to Montgomery representation is not needed. We use points in projective coordinates $(X{:}Y{:}Z)$, and, for any $\lambda \neq 0$, $(\lambda X{:}\lambda Y{:}\lambda Z)$ is another valid representation of the same point. Thus, when

Listing 10: Bytecode: main loop and exit sequence.

```
        # Check that the point in (Qx,Qy) is a valid curve point.
        _CALL check_point

        # Compute W = u*G + v*Q
        # Set work point W to the neutral (0:1:0). Values Wx and Wz were
        # initialized at zero by the interpreter initializatio, so we only
        # need to set Wy to 1.
        _LD _ONE ; _ST _Wy
        # We loop over bits in u and v (high to low).
        _FOR
        # Double W.
        _LD _Wx ; _ST _Hx ; _LD _Wy ; _ST _Hy ; _LD _Wz ; _ST _Hz
        _CALL point_add_to_W
        # Add G if bit i of u is set; u is in Ke
        _LD _Gx ; _ST _Hx ; _LD _Gy ; _ST _Hy ; _LD _ONE ; _ST _Hz
        _SKIPBITZ _Ke ; _CALL point_add_to_W
        # Add Q if bit i of v is set; v is in Ks; Hz still contains 1
        _LD _Qx ; _ST _Hx ; _LD _Qy ; _ST _Hy
        _SKIPBITZ _Ks ; _CALL point_add_to_W
        # Loop until all bits have been processed.
        _NEXT

        # Get x coordinate of point W, canonicalized. For now we suppose
        # the W is not the point-at-infinity.
        _LD _Wz ; _CALL invert_mod ; _MUL _Wx ; _CALL canonicalize

        # Signature is valid if and only if we get back the same value
        # as r, when reduced modulo n. We switch back to computations
        # modulo n, then compute r - x(W) mod n.
        _MODN ; _SUB _Kr
        # To compare with zero, we reduce the value; this entails an
        # extra division by R, but this does not matter since we are
        # only interested in knowing if the value is zero.
        _MRED ; _SKIPNZ ; _OK ; _FAIL

        # If W was the point-at-infinity, then inversion of Wz yielded zero,
        # and the rebuilt x(W) was zero. Since r != 0 (we checked it
        # explicitly), this case implies a signature rejection, which is
        # the correct result.
```

Listing 11: Bytecode: curve point addition (projective, complete).

```
# Add a curve point to point W. Point W is in projective coordinates
# in (Wx:Wy:Wz); the other operand is read from (Hx:Hy:Hz).
# Inputs:
#    Wx, Wy, Wz   first operand coordinates
#    Hx, Hy, Hz   second operand coordinates
# Output is written back into (Wx:Wy:Wz).
# Clobbers: T0 to T8
point_add_to_W:
        # t0 <- X1 * X2
        _LD _Wx ; _MUL _Hx ; _ST _T0
        # t1 <- Y1 * Y2
        _LD _Wy ; _MUL _Hy ; _ST _T1
        # t2 <- Z1 * Z2
        # t7 <- bb * t2
        _LD _Wz ; _MUL _Hz ; _ST _T2 ; _MUL _Bm ; _ST _T7

        # t3 <- (X1 + Y1)*(X2 + Y2) - t0 - t1
        _LD _Wx ; _ADD _Wy ; _ST _T3
        _LD _Hx ; _ADD _Hy ; _MUL _T3 ; _SUB _T0 ; _SUB _T1 ; _ST _T3

        # t6 <- (X1 + Z1)*(X2 + Z2) - t0 - t2
        _LD _Wx ; _ADD _Wz ; _ST _T6
        _LD _Hx ; _ADD _Hz ; _MUL _T6 ; _SUB _T0 ; _SUB _T2 ; _ST _T6
        # t6 is still in acc
        # t5 <- t1 + 3*(t6 - t7)
        # t7 <- t1 - 3*(t6 - t7)
        _SUB _T7 ; _ST _T5 ; _ADD 0 ; _ADD _T5   # acc <- 3*(t6 - t7)
        _ST _T7 ; _ADD _T1 ; _ST _T5
        _LD _T1 ; _SUB _T7 ; _ST _T7

        # t8 <- 3*t2
        _LD _T2 ; _ADD 0 ; _ADD _T2 ; _ST _T8

        # t6 <- 3*(b*t6 - t0 - t8)
        _LD _T6 ; _MUL _Bm ; _SUB _T0 ; _SUB _T8 ; _ST _T6
        _ADD 0 ; _ADD _T6 ; _ST _T6

        # t4 <- (Y1 + Z1)*(Y2 + Z2) - t1 - t2
        _LD _Wy ; _ADD _Wz ; _ST _T4
        _LD _Hy ; _ADD _Hz ; _MUL _T4 ; _SUB _T1 ; _SUB _T2 ; _ST _T4
        # t4 is still in acc
        # X1 <- t3*t5 - t4*t6
        _MUL _T6 ; _ST _Wx
        _LD _T3 ; _MUL _T5 ; _SUB _Wx ; _ST _Wx

        # t0 <- 3*t0 - t8
        _LD _T0 ; _ADD 0 ; _ADD _T0 ; _SUB _T8 ; _ST _T0
        # t0 is still in acc
        # Y1 <- t0*t6 + t5*t7
        _MUL _T6 ; _ST _Wy ; _LD _T5 ; _MUL _T7 ; _ADD _Wy ; _ST _Wy

        # Z1 <- t4*t7 + t3*t0
        _LD _T4 ; _MUL _T7 ; _ST _Wz
        _LD _T3 ; _MUL _T0 ; _ADD _Wz ; _ST _Wz

        _RET
```

we obtain affine, non-Montgomery coordinates (x, y) (for point Q or for point G), then the $(x{:}y{:}1)$ projective coordinates are also $(\mathcal{M}(x/R) : \mathcal{M}(y/R) : \mathcal{M}(1/R))$, which designate the same point, and are all in Montgomery representations. In other words, the use of projective coordinates gets us a conversion to Montgomery representation for free. This is also why we converted G_x and G_y (the generator coordinates) *out* of Montgomery representation (in listing 8, lines 11-14): we needed to complete the affine coordinates into projective coordinates with the proper Z, but we do not have $\mathcal{M}(1)$, only 1 inside the VM.

At the end of the computation, we obtain $\mathcal{M}(x_W)$ but canonicalization gets us the plain, non-Montgomery value, as befits ECDSA.

4.4 Assembly Optimizations

The virtual CPU must still be implemented, which is a matter of local assembly optimization. Conceptually, the emulator is just a loop which reads the next instruction byte, then jumps to the code sequence that handles specifically that byte. The instruction encoding was designed to support a two-level approach:

- A first dispatch is done on the low three bits of the instruction (which uses a lookup table with 8 entries). Instructions with a numeric parameter (e.g. `_LD` or `_MUL`) store that parameter in the five high bits of the instruction, so they are directly handled that way. The address in RAM of the variable pointed to by that parameter is systematically computed into a dedicated register prior to the dispatch.
- Parameter-less instructions (e.g. `_NORM` or `_FAIL`) all have their three low bits equal to 111 (in binary): the corresponding first-level dispatch routine then does a secondary dispatch, based on the five higher bits.

Generic Approach. The main general approach is to define a custom call convention. The instruction-processing routines called by the virtual CPU dispatch are functions, but they do not need to follow the conventional ABI with caller-saved and callee-saved registers: these functions are never called from the outside. *State invariants* such as proper stack alignment must be maintained (to support asynchronous code execution, i.e. "signals"), but we can define the role and preservation of any register as we see fit. In particular:

- Some dedicated registers should be reserved, if possible, for maintaining the instruction pointer, the state pointer (where variables are located), the current loop context (loop iteration point and counter), and the current modulus. Note that the modulus is not only its value as an integer, but also the Montgomery coefficient $g = -1/m \bmod^+ 2^s$, which we store in pre-shifted format after the modulus limbs (this is why the modulus is handled as a special variable and has dedicated modulus-switching instructions).
- Routines may share some code. Even if each instruction routine is invoked as a function, it may "tail-call" into another routine with a jump, or even by "falling-through" into that other routine if it is located immediately afterwards in the code space. In our implementations, for instance, all three "skip" instructions share the same sequence to enact the skip, if it was decided that it must be performed.
- The native stack is used to support the `_CALL` and `_RET` opcodes. This implies a non-standard stack management, which has a tendency to make debugging somewhat obscure (we do not maintain frame pointers, so the debugger cannot print stack traces).

The input values ultimately are five big integers, from three distinct sources (public key, signature, hash value), that must be decoded into limb sequences. A dedicated function (in our code, we call it `decode_int()`) should be used and invoked repeatedly; we also use it for the hardcoded constants (curve generator coordinates, moduli p and n), which allows us to store these constants in a compact 32-byte format, rather than the 40 bytes for five limbs. For some architectures, the opcode for a local function call is somewhat large; e.g. on x86 in 64-bit mode, a `call` to a function by name yields a 5-byte encoding. In these cases, it is beneficial to first store the address of the `decode_int()` function in a register, then do the call with register dereferencing (2-byte encoding per call on x86).

In our implementations, we define the virtual CPU loop as three macros (initialization, main loop, exit sequence) so that extra functions may be defined (with conditional compilation) for unit tests that invoke bytecode sequences. A comprehensive set of such unit tests, in particular for Montgomery multiplications, inversions, and curve point additions, is highly recommended.

x86 Optimizations. The x86 architecture has a very long history, starting with the Intel 8086, first released in 1978, and already inheriting some design elements from the earlier 8080 (this was made to allow easier conversion of existing 8080 code to the new platform). This implies a very complicated instruction encoding, with various optional prefixes that have been added along the years. The main consequence is that not all registers are equivalent, some yielding substantially shorter instruction encodings. The main elements to remember are the following:

- The "historical registers" (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, and `esp`) are usually better; e.g. `addl %eax, %ebx` (addition in AT&T syntax) is encoded over two bytes, but `addl %r8d, %ebx`, which uses the "new" register `r8d`, needs three bytes. Some specific operations (in particular `and` and `test`) have a further shortened encoding when the first operand is `eax` (or its low byte `al`).
- 64-bit operations need an extra prefix, thus 8-bit or 32-bit operations should be preferred when possible.
- Immediate constants that do not fit in $[-128, 127]$ are usually encoded over four bytes, and should be avoided. This also applies to offsets in address calculations, and to jumps (jumps to nearby locations have a smaller encoding).
- 32-bit operations automatically zero-extend the destination register to 64 bits. To set a register to zero, use `xorl` (32-bit bitwise `xor`) on the register with itself. This is compact (2-byte encoding) and is also semi-officially "blessed" (this sequence is recognized by the CPU instruction decoder, and handled in the register renaming unit, for an approximately null runtime cost).
- "Old" instructions have the shortest encodings. This includes instructions such as `loop` (decrement a counter and jump if non-zero) and `xlat` (table lookup with a 1-byte index). These instructions are often slow (microcoded execution) and have strict constraints on the involved registers, which impacts register allocation.
- Some instructions perform a memory access *and* an index update. These are in particular the `lods`, `stos` and `movs` families of instructions. Judicious use of these instructions can help reduce code footprint, but they have strict requirements on register allocation (e.g. `lods` always uses `rsi` as index, and loads the value into `eax`/`rax`).

– `push` and `pop` are encoded over a single byte each when the involved register is an historical one, and move 64 bits at a time. To copy a 64-bit value from an historical register to another one (e.g. `rcx` to `rdi`), a `push`/`pop` combination is two bytes, while a 64-bit `mov` is three bytes. Note that the `mov` is faster, since it is usually handled by the register renaming unit, and involves no memory traffic (the CPU will normally do the memory write of the `push` asynchronously, and optimize the `pop`, but the memory write will still need to happen at some point, which is not free).

RISC-V Optimizations. RISC-V defines core integer instructions (I) and many sets of extra optional instructions; any hardware implementation may include some of these extra instruction sets, and formal *profiles* are defined which list instruction sets whose presence is guaranteed. For our ECDSA implementation, we use the 64-bit mode with the core integer instructions (I) as well as the multiplication instructions (M — only multiplication instructions, not division), and the compressed instruction encodings (C). These instruction sets are part of all currently defined "application-processor" profiles.

RISC-V instructions are simple, with a regular encoding; most of the size optimization efforts are in register allocation to benefit from the compressed instruction encodings (16-bit instructions instead of 32-bit): while the CPU has 31 general-purpose registers which are mostly interchangeable, most compressed encodings can be used only with the 8 "popular" registers (`s0`, `s1`, and `a0` to `a5`). To achieve a low code size, the most used registers should be chosen among these registers.

Another noteworthy point is that the local function call instruction (`jal`) with the target being designated by a label uses a 32-bit encoding, while the call-to-register instruction (`jalr`) can use a compressed encoding. Just like on x86, we leverage the shorter `jalr` by first obtaining the address of the `decode_int()` function into a register, then using it repeatedly with `jalr`. Similarly, getting the address of a function or a static variable into a register uses the `lla` sequence, which consists of two 32-bit instructions; we endeavour to have as few such sequences as possible, hence we ensure that `decode_int()`, the block of constants, and the bytecode entry point are all consecutive in the address space, so that a single `lla` may be shared for all three elements.

Armv8-A Optimizations. Armv8-A instructions all use 32 bits; contrary to RISC-V (and to the "Thumb" instructions on 32-bit Arm profiles), there are no compressed 16-bit instructions, thus little to gain there. This simplifies register allocation, since general-purpose registers are interchangeable. The local function call (`bl`) uses a single 32-bit instruction; so does the `adr` instruction, which loads the address of a function or a static variable in a register. There is therefore no need to get the address of `decode_int()` in a register, or to locate the block of constants immediately after that function.

Most size optimization efforts on Armv8-A are spent trying to leverage dual-operation instructions, such as memory accesses that also update the index register, or arithmetic and logic instructions where an extra shift or rotation can be applied to the second operand. In total, Armv8-A offers fewer opportunities for size optimizations than RISC-V and in particular x86[11].

[11] To some extent, this why I find assembly programming on Armv8-A more pleasant than on x86.

4.5 Missing Optimizations

A few extra optimizations are possible on the implementations, but I refrained from applying them because they would have a substantial cost on performance, or, more importantly, would impact the implementation clarity and make it more difficult to run unit tests and validate the correctness of the implementation.

The virtual CPU runs a loop with a conditional jump over an exit flag, and there is a separate macro that winds down the context and translates the exit status back into the convention expected by the API function caller. These steps could be merged into the `_OK`/`_FAIL` code sequence, which would save a few bytes; but it would make it quite harder to write unit tests that exercise individual bytecode sequences, unless the unit test functions are made to use a different interpreter, which would remove most of the point of running unit tests (such tests have value precisely because they exercise parts of the implementation, and not test-specific code).

Current `_CALL` implementation is generic and supports arbitrary nesting levels of `_CALL` and `_RET`. The actual implementation only uses one level, and could use a dedicated register to remember the caller's instruction pointer, instead of using the native stack; this would make implementation of `_CALL` and `_RET` a bit shorter, and would allow using the native stack pointer to reference the interpreter context. However, that would prevent some *potential* improvements in the bytecode in which two-level calls would be useful (e.g. to merge sub-sequences of code in `point_add_to_W`).

The current `point_add_to_W` was obtained by directly translating the Renes-Costello-Batina algorithm into 43 three-instruction sequences, then doing manual optimizations. It is unlikely that the current state is optimal in that respect; further inspection of the operations would probably reveal a few opportunities for saving some more bytes.

The current modulus (p or n) is currently set with the `_SMOD` instruction, and read accesses as variable `_M` are supported by some pointer-swapping in instruction decoding. This is done so because a modulus is not only an integer but also includes the Montgomery coefficient $2^{w-s}(-1/m \mod^{+} 2^s)$, which is located immediately after the modulus in RAM. This special treatment could be avoided by making variable `_M` a normal value, but expanding all values to an internal format with an extra slot for the Montgomery coefficient; this slot would simply follow along in value copies, and would be otherwise ignored except when the value is used as the modulus in Montgomery multiplications.

Presumably, unleashing an AI on our current code could make it somewhat shorter, but readability and correctness validation would be lost in the process. This validation is a crucial element for cryptographic implementations; this is covered in the next section.

5 Comprehensive Range Analysis

The description of the ECDSA implementation in section 4 highlighted that the correctness of the implementation relies on the following properties:

- All the additions and subtractions produce the correct integer results, i.e. on inputs that represent integers x and y, addition returns an output that represents $x + y$, and subtraction similarly returns $x - y$. This is not prescriptive on the actual representation (in redundant representation, the same integer may accept multiple representations).
- Montgomery multiplications return, for inputs x and y, and current modulus m, an output z such that $z \equiv xy/R\ [m]$. This is again not prescriptive on representations: it does not require a specific z value since only modular equality is needed here.
- Integer canonicalization properly returns an integer in the $[0, m-1]$ range, for the modulus m. This property is required for proper bit enumeration over the scalars u and v for the double point multiplication $uG + vQ$, as well as for correctly handling the reinterpretation of x_W into a scalar.

This section is about proving these properties for any possible inputs to the ECDSA verification algorithm. This is not a *complete* formal proof of the entire implementation, in that it does not cover some needed tests, such as verifying that the input r and s are in the $[1, n-1]$ range, or the public key point is indeed a curve point. Such properties are amenable to a manual, human-based verification by simply looking at the code; contrary to rare mathematical edge cases, notably related to carry propagation, their analysis should be simple. You can add some explicit test vectors if you think that your developers would manage to goof up something as simple as, for instance, verifying that the input hash value length is between 32 and 64 bytes[12].

The proving operation is done in a Python script called `rr_ecdsa_range.py` and provided with the ECDSA implementations.

5.1 Range Tracking

The first two properties boil down to verifying that none of the operations, which we expressed in section 2 as functions `add()`, `sub()`, `longmul_s()`, etc, overflows its expected container; we check this no-overflow property by doing a range analysis which keeps track of possible values as ranges. We define a *range* as a pair of integers (L, H) which mean that the value is an element of the interval $[L, H]$ (note that both ends are included). When $L = H$, the range is said to be *exact* since it pinpoints a single value.

In `rr_ecdsa_range.py`, the `ZRange` class is defined; each instance is immutable and represents a range. Elementary operations on ranges are supported, such as additions, multiplications, etc; for instance, when multiplying range (L_1, H_1) with range (L_2, H_2), the output range is (L_3, H_3) which is such that L_3 and H_3 are, respectively, the minimum and the maximum in the quadruplet $\{L_1L_2, L_1H_2, H_1L_2, H_1H_2\}$. We also need to support masking values to their t low bits: signed and unsigned t-bit masking of an integer x mean returning $x \bmod^{\pm} 2^t$ and $x \bmod^{+} 2^t$, respectively. For ranges, the implementation of unsigned masking returns the full range $[0, 2^t - 1]$ if the input $[L, H]$ is such that $H - L \geq 2^t$ (all low t-bit

[12] My unit test framework has such tests because I did goof it up at some point.

patterns are possible) or if $H \mod^+ 2^t < L \mod^+ 2^t$ (which implies that both 0 and $2^t - 1$ are possible, thus encompassing the whole range, even if some of the values in the range are not actually possible); otherwise, the relevant sub-range of $[0, 2^t - 1]$ is returned. Signed masking uses unsigned masking with an appropriate offset applied before and after the operation.

Building on `ZRange`, the `ZWord` class represents a machine word of w bits. Each instance encapsulates two `ZRange` instances: one represents the possible range for the value stored in the word, and the other is the container range, which is either $[-2^{w-1}, 2^{w-1} - 1]$ (the word is interpreted as a signed integer) or $[0, 2^w - 1]$ (the word is interpreted as an unsigned integer). Arithmetic operations are implemented on the value range, and automatically raise an exception if the output value range would not entirely fit within the container range: this is how overflows on word operations are tracked. Explicit "wrapping" operations are also provided, which truncate the output as appropriate instead of raising an exception; this is used for the computation of the Montgomery factors f_i, for which the truncation, and reinterpretation of an unsigned value as a signed integer, are part of the operation semantics.

The `MLInt` class builds on `ZRange` and `ZWord` to implement a multi-limb redundant representation of an integer. Each `MLInt` instance has three fields:

- A reference to an overall context structure which contains parameters such as k (number of limbs), s (for radix 2^s) or flags such as whether signed or unsigned normalization should follow a Montgomery multiplication.
- The integer limbs, as an array of k instances of `ZWord` (with signed interpretation): we keep track of the individual ranges of all limbs separately.
- An overall integer range (as a `ZRange`), which qualifies the represented big integer. This may be a strict sub-range of the possible values that could be inferred from the ranges of the individual limbs.

Again, addition, subtraction and negation are implemented, as well as Montgomery multiplications. This last operation uses a modulus as extra parameter, provided as a `MLIntModulus` instance, which contains the modulus limbs — as `ZWord` values, which are not necessarily exact if we want to do range analysis with moduli which are not known at compile-time — and some extra parameters such as the Montgomery coefficient $-1/m \mod^+ 2^s$.

All the `ZRange`, `ZWord` and `MLInt` instances are immutable; all mathematical operations return new instances with the result. Using immutable instances increases memory allocation traffic (hence garbage collection costs) but avoids all possible issues with concurrent modifications to the same object.

Since `ZWord` instances automatically check for overflows, executing the operations without any thrown exception implies a correct result for all input values fitting in the provided input ranges.

5.2 Application to ECDSA Verification

We can use `MLInt` in a full ECDSA verification: we set the input ranges for signature elements, public key coordinates, and hash value to the full 256-bit range $[0, 2^{256} - 1]$, and then run the algorithm to completion. This requires a representation of the ECDSA code as a sequence of operations that the Python script can map to operations on `MLInt` instances. But we already have such a representation: the bytecode described in section 4.

The `rr_ecdsa_range.py` thus includes a textual copy of the bytecode (as found in the assembly implementations) coupled with a minimal parser that turns it into an internal representation of the bytecode instructions; a bytecode interpreter then executes the instruction just like the virtual machine assembly implementations, except that it maintains the state variables as `MLInt` instances. Two API functions are provided:

- `test_ecdsa()` uses exact ranges as inputs (signature, public key, hash value), i.e. ranges that pinpoint a single value, obtained from a test vector. All 492 Wycheproof test vectors are used. Since the input values are exact, all derived values in the bytecode execution are also exact, and this just runs the ECDSA verifier. Going through all the test vectors is meant to confirm that the bytecode interpreter in `rr_ecdsa_range.py` is properly implemented[13].
- `range_analysis()` instead sets inputs to the full 256-bit range. As the code unfolds, maximum possible ranges for all words and big integers are computed at all points of the execution; if the end of the computation is reached (the final `_OK` or `_FAIL`) without triggering any overflow check, then this proves that no overflow is indeed possible in the implementation. Additionally, extra tests are hooked on calls to the `canonicalize` routine, to check that the obtained big integer value range is indeed $[0, m - 1]$ for the relevant modulus m (n or p, depending on the location of the call).

One non-trivial point about this range analysis must be explained: the executed code sequence does not always run the same list of instructions, since conditional execution is used through the three provided "skip" instructions (`_SKIPBITZ`, `_SKIPNZ` and `_SKIPNEG`). When the tested value is known with an exact range, or at least with a range that is precise enough to yield the information (e.g. for `_SKIPNEG`, if the top limb range is entirely negative or entirely non-negative), then the interpreter can apply the skip semantics; otherwise, the skip operation is said to be *unresolved* and the interpreter must assume that both behaviours (skip and non-skip) are possible, run *both*, and merge the resulting states. Two `MLInt` values are merged by pairwise merging the ranges of their limbs and overall values, the merge of two ranges being the smallest range that includes both source ranges. This process must handle a few situations:

- When the skip operation is over a `_CALL` instruction, the called function must be invoked with a fresh copy of the VM state, so that after the function execution, the output state resulting from the function invocation can be merged into the saved state.
- Unresolved skips over `_OK` or `_FAIL` are simply ignored, since then the no-skip branch exits the VM and contributes no state to merge.
- To handle conditional returns in functions (unresolved skips over `_RET`), the call stack records not only the caller's instruction pointer, but also a slot for a "merged state" which corresponds to the output states of all conditional returns. This allows computing the overall output state of the function, for all possible ways the function may exit.
- For `_SKIPNEG`, the two branches are executed over states that take into account the result of the test: the skipped instruction uses a state that includes the information that the top limb of the value in the accumulator was non-negative (the overall integer value range is

[13] Running all 492 tests can take a lot of time, more than one hour for some choices of k and s, due to the inherent inefficiency of keeping track of ranges and allocating all these values.

also adjusted accordingly), while the saved state (for the "instruction skipped" branch) includes the information that the top limb of the value in the accumulator was negative. This is needed in order to prove that canonicalization works properly.

– If the skip instruction applies to `_NEXT`, `_FNEXT`, or another skip instruction, then this is explicitly detected and an exception is thrown. This case does not happen in the ECDSA verifier bytecode. Supporting these cases would complicate the range analysis, since the simple execute-and-merge process would no longer suffice.

The `test_ecdsa()` and `range_analysis()` functions accept the same parameters, the first three of which being mandatory:

`k` Number of limbs (integer).

`s` Limb radix size, in bits (integer).

`w` Machine word size, in bits (integer).

`slf` If `True`, then the Montgomery factors use signed interpretation (a `longmul_s` is used for multiplying f_i with modulus limb m_j, as shown in listing 2). For the integer canonicalization, as implemented in listing 5, to work properly, `slf` must be set to `True`. Default value is `False`.

`slv` If `True`, then signed normalization is applied at the end of Montgomery multiplication; default value is `False`, which means that unsigned normalization is applied (as in listing 2).

`slm` If `True`, then the moduli (p and n) are stored and used in signed-normalized representation; default value is `False`, which implies that the moduli are unsigned-normalized.

`spec` A set of flags, as an integer which is a bitwise OR of individual flags (flag j has numerical value 2^j). When set, each flag adds a `_NORM` instruction at a specific place; these extra unsigned normalization instructions are needed in some cases, depending on the values of the other parameters (this will be detailed below). Default value is 0 (all flags cleared).

A sample usage in a Python command-line prompt is the following:

```
$ python3
Python 3.12.3 (main, Mar  3 2026, 12:15:18) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from rr_ecdsa_range import *
>>> range_analysis(5, 54, 64, slf=True)
OK
>>> test_ecdsa(5, 54, 64, slf=True)
.............................. (etc)
```

The fact that `range_analysis()` prints "OK" and exits without raising an exception shows that when $k = 5$, $s = 54$ (54-bit limbs), $w = 64$ (64-bit limbs), and Montgomery factors f_i are interpreted as signed values, then the entire ECDSA implementation never overflows for any input. Running `test_ecdsa()` furthermore shows that the computation is correct enough for all 492 test vectors.

5.3 Variants and Flags

The effect of the `slf`, `slv`, `slm` and `spec` parameters is best explained with examples. If we try the calls above without `slf=True`, then Montgomery factors are unsigned and the calls fail. For instance:

```
>>> range_analysis(5, 54, 64)
<current IP = 8 [line 14]: ST 5>
(...)
    assert x.low >= 0 and x.high <= mod_N - 1
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

The designated instruction is the `_ST` that immediately follows a call to `canonicalize` (with modulus n): the canonicalization relies on Montgomery reduction to return a value in $[-\gamma n, +\gamma n]$ for some constant $\gamma < 1$, and this is ensured only if the Montgomery factors are signed; otherwise, the overall reduction factor f can become greater than $R/2$, and on output of Montgomery reduction, one may obtain an integer equal to or slightly greater than the modulus, which is not correctly handled by the current implementation of `canonicalize`, which does only one conditional *addition* of the modulus, but not a conditional *subtraction*. All our implementations thus use `slf=True`.

The $(k, s, w) = (5, 54, 64)$ setting with `slf=True` is what we implement in the 64-bit C code (`gen64`) and all three 64-bit assembly implementations (`amd64`, `arm64` and `rv64`).

If we set `slm=True`, then range analysis still passes successfully, but unit tests fail:

```
>>> range_analysis(5, 54, 64, slf=True, slm=True)
OK
>>> test_ecdsa(5, 54, 64, slf=True, slm=True)
(...)
    assert status == 'F'
           ^^^^^^^^^^^^
AssertionError
```

The effect of setting `slm` to `True` is that the moduli are internally signed-normalized instead of unsigned-normalized. For instance, in signed-normalized format, the two low limbs of p are -1 and 2^{42}, instead of $2^{54} - 1$ and $2^{42} - 1$, respectively. Signed normalization of the modulus has the effect that when adding $f_i m$ to the intermediate result in Montgomery representation, the integer products $f_i m_j$ are smaller in absolute value, which can help with avoiding an integer overflow in the limbs. The effect is usually marginal, but with some combinations of k, s and w, it can potentially help succeed through a sequence of operations that would otherwise potentially trigger an overflow. Unfortunately, using a signed-normalized modulus breaks the `invert_mod` implementation (listing 6) because the FLT code needs the exponent $m - 2$ in canonicalized (unsigned-normalized) format and expects to obtain it by simply subtracting 2 from the low limb, which does not work if m is signed-normalized. The solution is to add an extra `_NORM` instruction to unsigned-normalize the exponent, which we can do by setting the special flag 0 through the `spec` parameter:

```
>>> test_ecdsa(5, 54, 64, slf=True, slm=True, spec=1)
............................... (etc)
```

This enlarges the bytecode by 1 byte.

If we try to make a 32-bit version, we can find that with only `slf=True`, range analysis finds no workable solution with $k \leq 12$. For instance:

```
>>> range_analysis(12, 22, 32, slf=True)
<current IP = 198 [line 217]: MUL 23>
<current IP = 54 [line 50]: SKIPBITZ 4>
(...)
Exception: value range [-2273312274, 1056964453] exceeds
container range [-2147483648, 2147483647]
```

This indicates that a limb potentially overflows. The printed reference to the offending instruction points to a `_MUL` opcode which is near the end of the `point_add_to_W` routine. We can then try to put a `_NORM` call right before this multiplication: unsigned normalization makes individual limbs smaller, and the extra `_NORM` will do that on the accumulator, which is one of the operands of the `_MUL`. In the bytecode embedded in `rr_ecdsa_range.py`, we placed such a `_NORM` instruction, enabled with special flag 2 (i.e. by setting `spec` to 4):

```
>>> range_analysis(12, 22, 32, slf=True, spec=4)
OK
>>> test_ecdsa(12, 22, 32, slf=True, spec=4)
............................... (etc)
```

This extra `_NORM` enlarges the bytecode by 1 byte. This is the setting used in the alternate x86 assembly implementation (`amd64alt`).

Another option for a 12-limb 32-bit implementation is to use signed normalization at the end of each Montgomery multiplication (with `slv=True`). Signed normalization makes output limbs smaller (in absolute value) and thus should allow more successive additions and subtractions before the next multiplication. However, this would predictably break canonicalization: the `canonicalize` routine (listing 5) uses Montgomery reduction (`_MRED`) then tests the sign of the output with `_SKIPNEG`, under the assumption that the top limb will contain the value sign, which is true for unsigned-normalized values, but not for signed-normalized values. When `slv=True`, we thus need an extra `_NORM` between the `_MRED` and the `_SKIPNEG`, which in `rr_ecdsa_range.py` is obtained by setting the special flag 1 (value 2 into `spec`):

```
>>> range_analysis(12, 22, 32, slf=True, slv=True, spec=2)
OK
>>> test_ecdsa(12, 22, 32, slf=True, slv=True, spec=2)
............................... (etc)
```

This specific setup is what is used in the 32-bit portable C implementation (`gen32`). In general, it is slightly less desirable than the previous option (with `slv=False` and an extra `_NORM` in `point_add_to_W`) because it means that the compiled code will contain instructions for *both* signed and unsigned normalization, instead of just unsigned normalization.

An intriguing third option for a 12-limb 32-bit implementation is to use a signed modulus (`slm=True`, and special flag 0 set). It fails range analysis, but test vectors pass successfully, which can only mean one of two things: either the range analysis is not tight enough, and its inherent approximations lead it to reject a valid solution; or the value overflow is in fact possible, but rare, and the test vectors do not exercise the faulty condition. Avoiding rare edge case is exactly why we are doing an extensive range analysis, and we must therefore reject that option until an actual correctness proof can be established.

6 Conclusion

In this paper, we described an implementation technique for Montgomery multiplications that avoids most carry propagations and should be desirable for computations on platforms for which carry propagation is expensive, in particular RISC-V CPUs. The use of redundant representations with signed limbs also opens the possibility of performing unreduced limb-wise additions and subtractions, which may yield non-negligible performance benefits in a speed-oriented implementation of relatively complex code sequences, as is common in schemes using elliptic curves. This should be applicable, in particular, to post-quantum schemes based on isogeny graphs.

An additional potential benefit of redundant representations is, for moduli with a special format, selection of an alternate representation that would allow more efficient operations; for instance, the parameters proposed for the 128-bit security level of FESTA[3], a post-quantum isogeny-based public key encryption scheme, use a modulus m such that $m + 1$ is a multiple of 2^{632}; in a classic unsigned and non-redundant representation, this leads to a sequence of low limbs with a large value. However, in a signed redundant representation, we could set the lowest limb to -1, and then quite a few limbs beyond it will have value 0. Multiplication by -1 is efficient, multiplication by 0 even more so. The extent of such optimizations depends on the modulus value, but might lead to non-negligible speed improvements. Best case seems to be moduli whose binary representation includes a very long sequence of ones, so that signed limbs may turn all these into zeros.

We illustrated the technique with an example of questionable usefulness (a codegolfed implementation of ECDSA signature verification over NIST curve P-256), but appropriate for highlighting how a comprehensive range analysis can be performed, as is required in order to use the unreduced limb-wise additions and subtractions. Python-based tooling can in particular pinpoint where overflows may occur, and help decide where extra normalization steps need to be placed. The proof of impossibility of overflows, combined with the removal of most carry propagation, should help in making correct implementations.

In the implementation process, we also estimated how reliable and useful the current crop of AIs can be for implementing cryptographic primitives. Verdict is: they produce results quickly, but be very cautious, they tend to focus on test vectors, which cannot guarantee correctness. You really should ask the AI to produce a companion verifiable proof of correctness.

Acknowledgements

Many thanks to Steve Weis, who provided access to the AI.

References

1. Accredited Standards Committee X9, *The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute, ANS X9.62, 2005.
2. Accredited Standards Committee X9, *Key Agreement and Key Transport Using Elliptic Curve Cryptography*, American National Standards Institute, ANS X9.63, 2017.
3. A. Basso, L. Maino and G. Pope, *FESTA: Fast Encryption from Supersingular Torsion Attacks*, https://ia.cr/2023/660
4. D. Bernstein, *Curve25519: new Diffie-Hellman speed records*, Public Key Cryptography - PKC 2006, Lecture Notes in Computer Science, vol. 3958, pp. 207-228, 2006.
5. 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.
6. E. Brickell, *A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography*, Advances in Cryptology - CRYPTO 1982, pp. 51-60, 1983
7. J. Colson, *A Short Account of Negativo-Affirmative Arithmetick*, Philosophical Transactions (1683-1775), vol. 34, pp. 161-173, 1726.
8. W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, vol. 22, issue 6, pp. 644-654, 1976.
9. J. Honkala, *On Number Systems with Negative Digits*, Annales Academiæ Scientiarum Fennicæ, Series A. I. Mathematica, vol. 14, issue 1, pp.149-156, 1989.
10. LEAN *Programming Language*, https://lean-lang.org/
11. A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996, https://cacr.uwaterloo.ca/hac/
12. P. Montgomery, *Modular Multiplication Without Trial Division*, Mathematics of Computation, vol. 44, issue 170, pp. 519-521, 1985.
13. Information Technology Laboratory, *Digital Signature Standard (DSS)*, National Institute of Standard and Technology, FIPS 186-5, 2023.
14. Information Technology Laboratory, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, National Institute of Standard and Technology, FIPS 203, 2024.
15. Information Technology Laboratory, *Module-Lattice-Based Digital Signature Standard*, National Institute of Standard and Technology, FIPS 204, 2024.
16. D. Phatak and I. Koren, *Hybrid Signed-Digit Number Systems: A Unified Framework for Redundant Number Representations With Bounded Carry Propagation Chains*, IEEE Transactions on Computers, vol. 43, issue 8, pp. 880-891, 1994.
17. T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte and Z. Zhang, *Falcon*, Technical report, National Institute of Standards and Technology, 2019, https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions
18. J. Renes, C. Costello and L. Batina, *Complete Addition Formulas for Prime Order Elliptic Curves*, Advances in Cryptology - EUROCRYPT 2016, Lecture Notes in Compute Science, vol. 9665, pp. 403-428, 2016. https://ia.cr/2015/1060
19. R. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, vol. 21, issue 2, pp. 120-126, 1978.
20. *Project Wycheproof*, https://github.com/C2SP/wycheproof