ZKSECURITY

Audit of Rust p256 Crate

Date: April 7th, 2025

Introduction

On April 7th 2025, NEAR requested zkSecurity to perform a security assessment of the Rust `p256` crate (https://crates.io/crates/p256). No major issues were found and the codebase was found to be thoroughly tested and well-architected.

In-Scope Components

The scope of the audit included:

- **Elliptic Curve**. The p256 elliptic curve group operations, including (de)serialization and exposed functionnalities relevant to NEAR.
- **ECDSA**. ECDSA-related code, focused on signature verification (as it is NEAR's main use case).
- **Field Arithmetic**. The internal scalar and base field implementations for the curve, including dependencies on the `crypto-bigint` crate (https://github.com/RustCrypto/crypto-bigint).

Out-of-Scope Components

As the usage mainly focused on signature verification on 32-bit machine, the following items were not the focus of this audit:

- **Signature generation**. The audit focused on signature verification, as this is the main use case for NEAR.
- **Constant-time code**. Due to the previous point, while we did look at the constant-time code to ensure correctness of implementation, we did not focus on ensuring that the code was indeed constant-time (as signature verification does not involve secret data).
- **64-bit specific code**. This is important as many algorithms are reimplemented for 32-bit and 64-bit machines, and thus the lack of bug in the 32-bit implementation might not necessarily mean that the 64-bit implementation is bug-free.
- **Unrelated primitives**. We did not look at ECDH or Hash-To-Curve implementations, as they are not used in NEAR.

Code Base Reviewed

Worthy of note: the audit focused on the last commit of the `p256` crate which was `32343a78f1522aa5bd856556f114053d4bb938e0` at the time of the audit and integrated changes like the move from `generic-array` to `hybrid-array` (https://github.com/RustCrypto/elliptic-curves/pull/1011) not used in NEAR's code. Although we assume that NEAR will update to the latest version.

Reference Integrations Provided by NEAR

In addition, we were provided the following PRs as example usage of NEAR:

- usage of the crate example - https://github.com/near/intents/pull/43/files
- example of other precompiles implemented in nearcore - https://github.com/near/nearcore/pull/9317

Overview Of The P-256 Crate

The `p256` crate implements the NIST P-256 elliptic curve along with a number of cryptographic schemes that make use of P-256 (ecdsa, ecdh, hash-to-curve). P-256 is a widely used elliptic curve defined over a prime field and with a prime order. We define the curve later in this document.

Organization Of The Code

The `p256` crate is part of the elliptic-curves repository and provides high-level APIs to access primitives (like ECDSA and ECDH) built on top of P-256 as well as low-level P-256 functionalities. The crate includes local code but also builds on top of many other dependencies from the same ecosystem of RustCrypto crates:

- `elliptic-curve` (https://github.com/RustCrypto/traits): it provides the basic `Curve` trait and the `PublicKey` struct.
- `crypto-bigint` (https://github.com/RustCrypto/crypto-bigint): it provides big integer implementations based on 32-bit or 64-bit machine words.
- `hybrid-array` (https://github.com/RustCrypto/hybrid-array): it provides the `Array` type with const generics.
- `primeorder` (https://github.com/RustCrypto/elliptic-curves): it implements the affine/projective point struct and point arithmetic (`add` and `double` operations). This crate provides some macros (e.g., `impl_mont_field_element`) to define `Field` and implement field arithmetic. The `p256` crate does not use macros for performance reasons.
- `ff` (https://github.com/zkcrypto/ff): some field arithmetic operations are pulled from this crate. For example, the square root functions.

The signature primitives rely on:

- `signatures` (https://github.com/RustCrypto/signatures): it provides the core structs (`VerifyingKey` and `Signature`) and functions (`verify_prehashed`) for signature verification in ECDSA.
- `signature` (https://github.com/RustCrypto/traits): it defines the basic traits (`Verifier` and `PrehashVerifier`) for signature verification.
- `sec1` (https://github.com/RustCrypto/formats): it implements the SEC1 (https://www.secg.org/sec1-v2.pdf) encoding format.

Note that, as mentioned in the introduction, we did not look into the generic-array crate (https://github.com/fizyk20/generic-array) as it is not used anymore in the latest commit of the `p256` crate. Furthermore, while we looked at parts of the subtle crate (https://github.com/dalek-cryptography/subtle) to understand its logic, we did not focus on ensuring that it indeed provides constant-time operations, as this was not the focus of the audit.

The P-256 Curve

The P-256 elliptic curve is a widely used elliptic curve defined over a prime field, and with a prime order. It is known under different names and specified in different standards: P-256 in NIST's SP 800-186 (https://csrc.nist.gov/pubs/sp/800/186/final), secp256r1 in SECG's SEC2v1 (https://www.secg.org/SEC2-Ver-1.0.pdf), prime256v1 in ANSI X9.62 (https://www.x9.org/).

The curve is defined by the short Weierstrass equation:

$$y^2 = x^3 + ax + b$$

over the finite field F_p, where:

- $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- $a = -3$
- $b = 41058363725152142129326129780047268409114441015993725554835256314039467401291$

The following is an implementation of the P-256 curve in SageMath:

```
p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
a = 0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc
b = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
G = (0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,
0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)
n = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
h = 0x1

# Create the base field
Fp = FiniteField(p)

# Define the elliptic curve E: y^2 = x^3 + ax + b over Fp
E = EllipticCurve(Fp, [a, b])

# Define the generator point G
G = E(G[0], G[1])
```

Some of these properties have implications over the performance and security for the implementations.

Optimize multiplication operations. The constant coefficient $a = -3$ allows performance optimization for point addition and doubling by avoiding multiplication operations related to the constant a.

Optimize modular arithmetic. The sparse structure of the prime field p can be represented as 32-bit limbs in little-endian format:

$$[ffffffff, ffffffff, ffffffff, 00000000, 00000000, 00000000, 00000001, ffffffff]$$

These simplified values in each limb allow for efficient modular arithmetic using bitwise operations during the reduction process.

Avoid small subgroup attack. When the cofactor `h` is 1, all points on the curve belong to a single subgroup of order n. As there are no small subgroups, extra checks against small subgroup attacks are unnecessary.

BigInt Implementation Overview

In order to take advantage of the architecture's native word size, the `crypto-bigint` crate uses a representation of integers based on machine words. This allows for efficient arithmetic operations on large integers.

To implement this, the crate defines the type `Word` (and wrapper type `Limb(Word)` as `u32` or `u64` depending on the Rust feature flag `target_pointer_width`.

A `Uint<LIMBS>` type is then defined generically as a fixed-size array of `Limb`s. For example, the p256 code will make use of `Uint<8>` (resp. `Uint<4>`) to represent 256-bit integers on 32-bit (resp. 64-bit) architectures.

Furthermore, limbs are ordered in a little-endian way, meaning the least-significant limb comes first (on the left). This allows for efficient serialization and deserialization of integers, as well as efficient arithmetic operations, without the need to reorder limbs on little-endian architecture (which makes up the majority of modern systems).

For example, any `u8` value `n` can be represented as `[n, 0, 0, 0, 0, 0, 0, 0]` on a 32-bit architecture.

Field Implementation

As seen above, the P-256 curve is defined over a prime field that we call the **base field** (implemented in the `field` submodule). The order of the curve (i.e. the number of points) is prime, producing a field we call the **scalar field** (implemented in the `scalar` submodule).

Both fields are of size 256-bit, and have similar implementations relying on the same `U256 = Uint<8>` type (on a 32-bit machine).

A notable difference is that:

- the base field implementation is based on the Montgomery reduction
- the scalar field implementation is based on Barrett reduction

As Montgomery reduction requires converting inputs to Montgomery form (and then back to a canonical form), it is generally best suited for applications that perform many multiplications in a row. Barrett reduction, on the other hand, is more efficient for applications that perform only a few multiplications. ECDSA signature verification follow the latter approach and thus Barrett reduction is used.

We also note that while the underlying implementation supports Karatsuba multiplication for larger integers (i.e. with at least 16 limbs), the P-256 implementation uses schoolbook multiplication in the integers.

In the next sections we explain the implementations based on Montgomery and Barrett reduction.

Montgomery Form and Montgomery Reduction

Montgomery reduction is explained at a low-level in section 14.3.2 on the Handbook of Applied Cryptography. Below, we provide a more understandable explanation of the algorithm and its implementation.

What Is The Montgomery Form?

For a modulus m and an element x modulo m, we can shift them with some known Montgomery value $R > m$ in the following way:

$$\overline{x} = x \cdot R \mod m$$

This is the **Montgomery form** of a value x!

Naturally, we can unshift to recover the value:

$$\overline{x} \cdot R^{-1} = x \mod m$$

Montgomery reduction does exactly this: it multiplies a value with the inverse of R and produces the result modulo m. As such, it should be immediately clear that Montgomery reduction is *at least* useful to convert a value in Montgomery form back to its canonical form.

> *Note that for the inverse of R modulo m to exist we need R and m to be coprime. In other words, we need $\gcd(m, R) = 1$.*

When we perform field operations (add and multiply) we constantly need to perform modular reductions (to reduce the result of an operation if it became larger than m). Modular reductions can get quite expensive. In fact, they are the most expensive part of handling field elements. For example, reducing a large number modulo a 256-bit prime involves multiple division and subtraction steps, which are computationally intensive.

On the other hand, it turns out that modular reductions in the Montgomery form can be faster to compute. For example, consider two elements in Montgomery form, $a \cdot R$ and $b \cdot R$, each 256 bits. When we multiply them, the result is $c = ab \cdot R^2$, a 512-bit value. Using Montgomery reduction, we can efficiently reduce c modulo m to obtain $d = c \cdot R^{-1} \mod m$. This result, $d = ab \cdot R \mod m$, is still in Montgomery form and ready for further operations.

> *Note that for addition things are much simpler, adding two values in Montgomery form still gets us a value in Montgomery form which can be easily reduced by subtracting m if needed. As such, Montgomery form is useful mainly for optimizing modular reductions after multiplications.*

It remains to show: why is modular reduction faster in Montgomery form, and how to perform that modular reduction.

Let's tl;dr the first question first: **why is it faster?** This is because the Montgomery value is usually chosen as $R = 2^n$ for n big enough such that $R > m$, which makes operations extremely straightforward and fast as we can use bitwise operations which are usually fast on hardware. Indeed, if you didn't know that yet, a left shift `<<` is a multiplication by 2 and a right shift `>>` is a division by 2.

Next, we explain how the Montgomery reduction works.

How Does The Montgomery Reduction Work?

Notice that we would like to perform the following division by R in the integer to simulate the multiplication with R^{-1}:

$$\overline{x}/R$$

This will only work if $\overline{x}$ is divisible by R in the integers. As most likely it is not, we need to find a different element of the same residue class that is divisible by R. That is, a k such that $\overline{x} + k{\cdot}m$ is divisible by R in the integers. Or in other words:

$$\overline{x} + k{\cdot}m = 0 \mod R$$

We can directly solve for k:

$$k = -m^{-1}{\cdot}\overline{x} \mod R$$

Finally, if this all makes sense, you now understand that we can calculate $\overline{x}R^{-1}$ modulo m using the following formula in the integers:

$$\frac{\overline{x} + (-m^{-1}{\cdot}\overline{x} \mod R){\cdot}m}{R}$$

This should give us a value that is equivalent to $\overline{x}R^{-1} \mod m$ but not necessarily in a standard representative in $[0, m)$.

In the bound analysis we will see that the above formula is in $[0, 2m)$, producing either exactly $\overline{x}R^{-1}$ (the "*canonical representative*") or $\overline{x}R^{-1} + m$ (which is easy to correct).

Here's a quick way to see that in SageMath:

```python
class Montgomery:
    def __init__(self, modulus, R):
        self.modulus = modulus
        self.R = R
        assert R > modulus
        assert gcd(modulus, R) == 1

    def to_montgomery(self, x):
        return (x * self.R) % self.modulus

    def from_montgomery(self, x):
        R_inv = inverse_mod(self.R, self.modulus)
        return (x * R_inv) % self.modulus

    def redc(self, x):
        assert x >= 0
        assert x < self.modulus * self.R

        # this can be precomputed
        inv_mod_R = inverse_mod(-self.modulus, self.R)

        # we find a representative that is divisible by R
        k =  (x * inv_mod_R) & (self.R - 1) # k = (x * -m^(-1)) % R
        nominator = x + k * self.modulus
        assert nominator % self.R == 0

        # nominator / R
        res = nominator >> self.R.log(2)

        # we correct the representative we found
        if res >= self.modulus:
            return res - self.modulus
        else:
            return res

# examples
M = Montgomery(modulus=13 * 17, R=2^8) # we use an RSA modulus in this example

# (notice that it works for 0 and 1 as well which are special cases)
for x in range(0, 100):
    assert M.redc(M.to_montgomery(x)) == x
```

Bound analysis

In Montgomery reduction, the $\overline{x}$ is usually the multiplication result of two field elements. Thus, we assume $\overline{x} < m \cdot m < R \cdot m$. Then we have

$$\frac{\overline{x} + (-m^{-1} \cdot \overline{x} \mod R) \cdot m}{R} < \frac{R \cdot m + R \cdot m}{R} = 2m$$

This shows that the above formula is within the range of $[0, 2m)$.

Word-by-Word Montgomery Reduction

In practice, Montgomery reduction is usually implemented in a word-by-word fashion. Rather than computing with large integers directly, the algorithm operates over fixed-size words—typically 32 or 64 bits wide—and

processes them one at a time. This enables the use of fast hardware operations like word multiplication, and allows division to be replaced with bit shifts, which are significantly more efficient.

Let the word (or base) be b (usually 2^{32} or 2^{64}), and suppose the modulus m is an n-word integer. Define $R = b^n$ and assume $gcd(R, m) = 1$. The goal is to compute the Montgomery reduction of a $2n$-word integer T (we assume $T < mR$), i.e., compute:

$$T \cdot R^{-1} \mod m$$

The idea is to reduce T by one word at a time until it becomes a n-word integer. Notice that

$$\begin{aligned} T \cdot R^{-1} \mod m &= T \cdot b^{-n} \mod m \\ &= ((T \cdot b^{-1}) \cdot b^{-1}) \cdots b^{-1} \mod m \\ &= ((T \cdot b^{-1} \mod m) \cdot b^{-1} \mod m) \cdots b^{-1} \mod m \end{aligned}$$

We can just divide T by b per round, and repeat n round to produce the final $T \cdot b^{-n}$. In each round, the limb size of T will decrease by one, then after n round T will become a n-word integer, which is close to the final target. Below is the process of the idea:

- Step 0. Initialize $T_0 \leftarrow T$.
- Step 1. Compute $T_1 \leftarrow \frac{T_0 + k_1 \cdot m}{b}$, where $k_1 = -m^{-1} \cdot T_0 \mod b$. This ensures that the numerator is divisible by b. Now, $T_1 = T_0 \cdot b^{-1} \mod m$, and T_1 is a $(2n-1)$-word integer.
- Step i (from 1 to n). Compute $T_i \leftarrow \frac{T_{i-1} + k_i \cdot m}{b}$, where $k_i = -m^{-1} \cdot T_{i-1} \mod b$. This reduces the word size of T_i by one in each step, ensuring $T_i = T_0 \cdot b^{-i} \mod m$.
- Final Step. After n iterations, T_n is a n-word integer equivalent to $T \cdot R^{-1} \mod m$. If $T_n \geq m$, perform a final subtraction: $T_n \leftarrow T_n - m$.

Note that the calculation of k_i can be simplified. First, precompute $m' = -m^{-1} \mod b$. Then, $T_i \mod b$ is simply the least significant limb of T_i, denoted as $T_i[0]$. Thus, k_i can be computed as:

$$k_i = m' \cdot T_i[0] \mod b$$

The overall algorithm proceeds as follows:

1. For each step i from 1 to n, reduce T_{i-1} by one word:

$$k_i \leftarrow T_{i-1}[0] \cdot m' \mod b$$

$$T_i \leftarrow \frac{T_{i-1} + k_i \cdot m}{b}$$

2. If $T_n \geq m$, then $T_n \leftarrow T_n - m$.
3. Output T_n.

This algorithm is efficient because it avoids costly division operations. Instead, it uses word-sized multiplication and addition, which are highly optimized on modern CPUs, making it well-suited for large integers.

Barrett Reduction

Barrett reduction is an algorithm to reduce a value modulo m. We have a value x that's in $[0, m^2)$ because a multiplication must have occured between two values modulo m. We would like to thus reduce the value modulo m.

Intuition on why it's fast

Barrett Reduction is a way to do that efficiently, relying on the same principles of the Montgomery reduction algorithm:

- we do things with limbs of size b the size of the machine words on which we're executing the algorithm
- we do mostly division and multiplication and modular operations with an operand that's a power of 2, because it's easy to do on our binary machines ($x/2^n$ translates to `x >> n`, $x \cdot 2^n$ is `x << n`, $x \mod 2^n$ is `x & (2^n - 1)`).

To make things even more optimal, we also define limbs using a base b corresponding to the machine word size (let's say $b = 2^{32}$ or $b = 2^{64}$) and k such that $b^k = \lceil log2(m) \rceil$. In other words the smallest k such that m fits in $32 \cdot k$ or $64 \cdot k$ bits.

Main idea

We want to compute $r = x \mod m$, which is the same as $x = q \cdot m + r$ for some q.

One technique to compute r is to compute q as:

$$q = \lfloor x/m \rfloor$$

And once you find q compute $r = x - q \cdot m$. Barrett Reduction allows us to compute q (or an approximation of q as you will see) more efficiently than with the formula above.

First, let's write the formula above as:

$$q = \lfloor x/m \rfloor = \left\lfloor \frac{x}{m} \cdot \frac{b^{2k}}{b^{2k}} \right\rfloor = \left\lfloor \frac{x}{m} \cdot \frac{b^{2k}}{b^{k+1} \cdot b^{k-1}} \right\rfloor = \left\lfloor \frac{b^{2k}}{m} \cdot \frac{x}{b^{k-1}} \cdot \frac{1}{b^{k+1}} \right\rfloor$$

This allows us to **precompute** $\mu = \frac{b^{2k}}{m}$ and to rewrite the above as:

$$q = \left\lfloor \frac{\frac{x}{b^{k-1}} \cdot \mu}{b^{k+1}} \right\rfloor$$

So far our computation is exact. But instead of computing q exactly, we'll approximate it by computing:

$$\tilde{q} = \left\lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \cdot \lfloor \mu \rfloor}{b^{k+1}} \right\rfloor$$

> *Note: Notice that $\lfloor \frac{\cdot}{b^{k-1}} \rfloor$ and $\lfloor \frac{\cdot}{b^{k+1}} \rfloor$ are fast to compute with right shifts.*

Since the two approximations can be smaller than their exact computations, we have that $\tilde{q} \leq q$. The analysis section will show that $\tilde{q} \in [q - 2, q]$.

Computing r in $x \approx \tilde{q} \cdot m + r$

If we had computed exactly q then what would be left for us to do would be to compute the result as:

$$r = x - q \cdot m$$

and since we know that $r \in [0, m)$ we also know that only the bitlength of m is involved in that subtraction. As such we can also compute it faster by only involving the least significant b^k bits:

$$r = x - q \cdot m \mod b^k = (x \mod b^k) - (q \cdot m \mod b^k)$$

(where modulo b^k can be computed efficiently)

But **remember, we have computed an approximation** $\tilde{q}$ of q, that is $\tilde{q} \in [q - 2, q]$.

We know that either:

1. $r = x - \tilde{q} \cdot m$
2. $r = x - (\tilde{q} + 1) \cdot m$
3. $r = x - (\tilde{q} + 2) \cdot m$

So what we do is that we compute 1 first, then if we don't get something smaller than m we subtract by m once or twice to get there.

This means that we might not have $\tilde{r} = x - \tilde{q} \cdot m < b^k$ right away. But instead a value m or $2m$ times larger.

Since $m < b^k$ we have that $2 \cdot m < 2 \cdot b^k$. This allows us to upperbound our approximation:

$$r \leq \tilde{r} \leq r + 2m < b^k + 2 \cdot b^k < b^{k+1}$$

Then we can calculate $\tilde{r}$ in a more efficient way:

$$\tilde{r} = ((x \bmod b^{k+1}) - (\tilde{q} \cdot m \bmod b^{k+1})) \bmod b^{k+1}$$

Again, here $\bmod b^{k+1}$ operation is very efficient on binary machine.

Analysis of the bound of Barrett reduction

Let's analyze the bound of calculated $\tilde{q}$ regarding to the actual quotient q, where

$$q = \lfloor \frac{x}{m} \rfloor$$

$$\tilde{q} = \lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \cdot \lfloor \frac{b^{2k}}{m} \rfloor}{b^{k+1}} \rfloor$$

First, since $\tilde{q}$ is derived from a truncated approximation of $\frac{x}{m}$, it naturally satisfies $\tilde{q} \leq q$.

Let's denote $\alpha = x \mod b^{k-1}$, where $\alpha < b^{k-1}$. Denote $\beta = b^{2k} \mod m$, where $\beta < m$. Then we can remove the floor operation:

$$\lfloor \frac{x}{b^{k-1}} \rfloor = \frac{x - \alpha}{b^{k-1}}$$

$$\lfloor \frac{b^{2k}}{m} \rfloor = \frac{b^{2k} - \beta}{m}$$

Then we can simplify $\tilde{q}$:

$$
\begin{aligned}
\tilde{q} &= \lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \cdot \lfloor \frac{b^{2k}}{m} \rfloor}{b^{k+1}} \rfloor \\
&= \lfloor \frac{\frac{x-\alpha}{b^{k-1}} \cdot \frac{b^{2k}-\beta}{m}}{b^{k+1}} \rfloor \\
&= \lfloor \frac{(x-\alpha)\cdot(b^{2k}-\beta)}{m \cdot b^{2k}} \rfloor \\
&= \lfloor \frac{x}{m} - \frac{\alpha \cdot b^{2k} + \beta \cdot (x-\alpha)}{m \cdot b^{2k}} \rfloor
\end{aligned}
$$

We use z to denote the red part (note that $z \geq 0$). Then we have $\tilde{q} = \lfloor \frac{x}{m} - z \rfloor$. By the floor function inequality $\lfloor x \rfloor + \lfloor y \rfloor + 1 \geq \lfloor x + y \rfloor$, we have:

$$
\lfloor \frac{x}{m} - z \rfloor + \lfloor z \rfloor + 1 \geq \lfloor (\frac{x}{m} - z) + (z) \rfloor = \lfloor \frac{x}{m} \rfloor = q
$$

That is saying $\tilde{q} + \lfloor z \rfloor + 1 \geq q$.

The bound of z is the key to analyze the bound of $\tilde{q}$. If we can prove that $0 \leq z < 2$, then we will have $\lfloor z \rfloor \leq 1$ and finally have $\tilde{q} + 2 \geq \tilde{q} + \lfloor z \rfloor + 1 \geq q$. That's exactly our goal. Let's analyze the bound of z.

Prove that $z < 2$

We have:

$$
\begin{aligned}
z &= \frac{\alpha \cdot b^{2k} + \beta \cdot (x-\alpha)}{m \cdot b^{2k}} \\
&< \frac{b^{k-1} \cdot b^{2k} + \beta \cdot b^{2k}}{m \cdot b^{2k}} \\
&= \frac{b^{k-1} + \beta}{m}
\end{aligned}
$$

We know that $b^{k-1} < m$ (because m is k-limb) and $\beta < m$. Then we have:

$$
z < \frac{b^{k-1} + \beta}{m} < \frac{m + m}{m} = 2
$$

And thus:

$$
\lfloor z \rfloor \leq 1
$$

That's exactly what we want. Therefore, we conclude that $\tilde{q} + 2 \geq \tilde{q} + \lfloor z \rfloor + 1 \geq q$.

Tighter bound: $z < 1$ in practice

We have proved that $z < 2$ under all circumstances. However, that is a loose bound. Here we claim that for almost all of the modulus m in practice, we have a tighter bound: $z < 1$ and thus have $\tilde{q} + 1 \geq \tilde{q} + \lfloor z \rfloor + 1 \geq q$.

Let's see what kind of m will have that tighter bound. Remembering that $z < \frac{b^{k-1}+\beta}{m}$, then $z < 1$ holds if the following holds:

$$
\frac{b^{k-1} + \beta}{m} \leq 1
$$

Which means:

$$b^{k-1} + \beta \leq m$$

Furtherly:

$$\beta \leq m - b^{k-1}$$

That's saying if β is in the range of $[0, m - b^{k-1}]$, then $z < 1$.

We know that $\beta = b^{2k} \mod m$. Then β is always in $[0, m)$. In practice, b is 2^{32} or 2^{64}. m is close to b^k. Therefore $m - b^{k-1}$ is very close to m. That means the range $[0, m - b^{k-1}]$ fills most of the range $[0, m)$.

Since $\beta \equiv b^{2k} \bmod m$ behaves like a uniformly random number in [0, m) for a random modulus m, it's highly likely that $\beta \leq m - b^{k-1}$. As a result, for most modulus m, we have $z < 1$ and thus $\tilde{q} + 1 \geq q$.

We assume that $\frac{b^k}{2} < m < b^k$ (which is the common case in practice) and β is uniformly random. Then

$$\Pr\left[\beta \leq m - b^{k-1}\right] = \frac{m - b^{k-1}}{m} > 1 - \frac{2}{b}$$

- If $b = 2^{32}$, then probability is greater than $1 - \frac{1}{2^{31}}$.
- If $b = 2^{64}$, then probability is greater than $1 - \frac{1}{2^{63}}$.

That is, **for nearly all moduli in practice**, the tighter bound $\tilde{q} + 1 \geq q$ holds.

The bound for P-256 scalar field

We have proven that, given a modulus m, if $\beta \leq m - b^{k-1}$ holds (where $\beta \equiv b^{2k} \bmod m$), then the tighter bound $\tilde{q} + 1 \geq q$ holds.

For the P-256 scalar field, it's not surprising that it satisfies $\beta \leq m - b^{k-1}$ for $b = 2^{32}$ or 2^{64}. This immediately implies the tighter bound:

$$\tilde{q} + 1 \geq q$$

That is, the computed $\tilde{q}$ is at most 1 less than the actual quotient q.

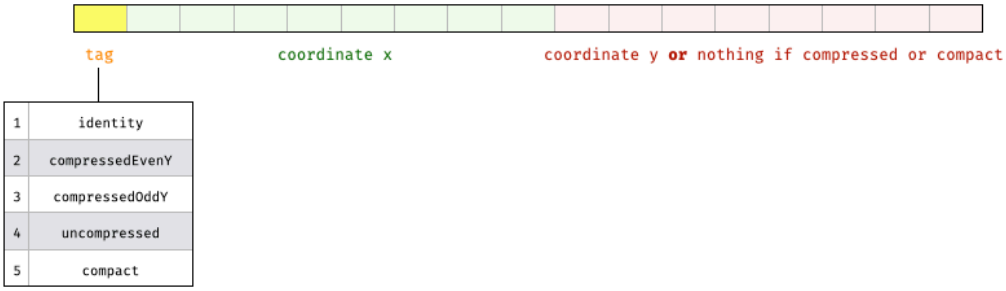
Therefore, the second `sub` here and here in the Barrett reduction—which conditionally subtracts another copy of the modulus—is unnecessary and can be safely removed in this case.

Elliptic Curve Implementation Detail

The `p256` module defines the basic properties of the curve shown in the P-256 section above, as well as the arithmetics for both base field and scalar field. The `primeorder` module provides the curve arithmetic specific for the curve property $a = -3$.

Encoding Points

The `primeorder` module uses the crate sec1 to define how the points on the curve should be encoded and decoded. Depending on the encoding tag, a point can be encoded in different formats: identity, compressed, uncompressed, and compact.



Identity (0x00) represents the point at infinity (or the zero point), which has no affine coordinates. The encoding only uses one byte, which is the tag `0x00`.

Compressed (0x02, 0x03) represents a point on the curve with only the x-coordinate provided. The y-coordinate is computed from the x-coordinate using the curve equation. Substituting the x-coordinate into the curve equation yields two possible y values, y and $-y$ resulted from the square root $y = \pm\sqrt{x^3 - 3x + b}$. Because the prime field p is odd, either $p - y$ or y is even, and the other is odd.

The tag `0x02` is used to choose even y-coordinates, while `0x03` is used to choose odd y-coordinates. Whether the y-coordinate is even or odd can be determined by taking $\pm y \mod 2$.

The total length of the compressed point is 33 bytes, which includes the tag byte and the x-coordinate (32 bytes).

Uncompressed (0x04) represents a point on the curve with both x and y-coordinates provided. The tag is followed by both the x-coordinate (32 bytes) and the y-coordinate (32 bytes). The total length of the uncompressed point is 65 bytes.

Compact (0x05) is similar to the compressed tags, but it always choose the larger y-coordinate resulted from the square root. This is not a encoding standard specified in SEC1 V2. But it is implemented in the crate dependency sec1.

Formulas In Affine Coordinates

To perform curve arithmetic we need to implement one main low-level operation: point addition, and sometimes a distinct point doubling operation as well. The latter is needed when the two points are identical because the vanilla formula for point addition would lead to a division by zero (unlike complete addition formulas when they are implemented).

When $Q \neq P$, the fomula for adding two distinct points:

$$
\begin{aligned}
\lambda &= \frac{y_2 - y_1}{x_2 - x_1}, \\
x_3 &= \lambda^2 - x_1 - x_2, \\
y_3 &= \lambda\,(x_1 - x_3) - y_1.
\end{aligned}
$$

This formula can't be applied to two identical points, because the denominator $x_2 - x_1$ in λ will be zero.

Therefore, when $Q = P$, the formula should be based on the tangent line of the curve at the point P, instead of the secant line between P and Q:

$$
\begin{aligned}
\lambda &= \frac{3x_1^2 + a}{2\,y_1}, \\
x_3 &= \lambda^2 - 2x_1, \\
y_3 &= \lambda\,(x_1 - x_3) - y_1.
\end{aligned}
$$

In affine form, these addition methods are sufficient to compute any point resulted from the scalar multiplication of a point, $[k]P$, on the curve.

Projective Formulas And Optimizations

The problem is that the division operation(realized as multiplying the inverse of one field) needed in λ is expensive and is harder for constant time computation. By converting the points to the projective form, these division operations can be replaced with multiplications and additions, which are more efficient, and easier to implement for constant time computing.

The implementation uses the algorithms from Renes-Costello-Batina 2015, which takes advantage of the fact that the constant $a = -3$, allowing further optimizations.

Point addition follows the Algorithm 4 from [RCB 2015], which expresses the result (X_3, Y_3, Z_3) in projective coordinates:

$$
\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)\left(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - b\,Z_1Z_2)\right) \\
&\quad - 3\,(Y_1Z_2 + Y_2Z_1)\left(b\,(X_1Z_2 + X_2Z_1) - X_1X_2 - 3\,Z_1Z_2\right), \\
Y_3 &= 3\left(3X_1X_2 - 3Z_1Z_2\right)\left(b\,(X_1Z_2 + X_2Z_1) - X_1X_2 - 3Z_1Z_2\right) \\
&\quad + \left(Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - b\,Z_1Z_2)\right)\left(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - b\,Z_1Z_2)\right), \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)\left(Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - b\,Z_1Z_2)\right) \\
&\quad + (X_1Y_2 + X_2Y_1)\left(3X_1X_2 - 3Z_1Z_2\right).
\end{aligned}
$$

The constants 3 come from the curve property $a = -3$. Because the constant 3 is small, instead of a full field multiply, the algorithm uses the Double and Add method to compute the multiplication by 3 with the intermediate results.

For example, it applies Double and Add method with the intermediate result $X_1Z_2 + X_2Z_1 - b\,Z_1Z_2$ for the effect of multiplication by 3, as shown in the code:

```
let bzz_part = xz_pairs - (C::EQUATION_B * zz); // 19, 20
let bzz3_part = bzz_part.double() + bzz_part; // 21, 22
```

For point doubling formula, the addition formula can be simplified to the following, because the points (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) are the same:

$$
\begin{aligned}
X_3 &= 2\,X\,Y\,\left(Y^2 + 3\,\left(2\,X\,Z - b\,Z^2\right)\right) \\
&\quad - 6\,Y\,Z\,\left(2\,b\,X\,Z - X^2 - 3\,Z^2\right), \\
Y_3 &= \left(Y^2 - 3\,\left(2\,X\,Z - b\,Z^2\right)\right)\,\left(Y^2 + 3\,\left(2\,X\,Z - b\,Z^2\right)\right) \\
&\quad + 3\,\left(3\,X^2 - 3\,Z^2\right)\,\left(2\,b\,X\,Z - X^2 - 3\,Z^2\right), \\
Z_3 &= 8\,Y^3\,Z.
\end{aligned}
$$

Similarly, instead of doing multiplications with the constants in red color, the implementation applies uses field addition operations with the intermediate results.

Mixed Addition

Mixed addition is another optimization method for point addition, in which one of the points is in affine coordinates and the other is in projective coordinates. By converting the point in affine coordinates to projective coordinates, the converted point has $Z_2 = 1$. By plugging it into the addition formula, it cancels the Z_2 in the formula:

$$
\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)\,\left(Y_1Y_2 + 3\big(X_1 + X_2Z_1 - b\,Z_1\big)\right) \\
&\quad - 3\,(Y_1 + Y_2Z_1)\,\left(b\,(X_1 + X_2Z_1) - X_1X_2 - 3\,Z_1\right), \\
Y_3 &= 3\,(3X_1X_2 - 3\,Z_1)\,\left(b\,(X_1 + X_2Z_1) - X_1X_2 - 3\,Z_1\right) \\
&\quad + \,(Y_1Y_2 - 3\,(X_1 + X_2Z_1 - b\,Z_1))\,(Y_1Y_2 + 3\,(X_1 + X_2Z_1 - b\,Z_1)), \\
Z_3 &= (Y_1 + Y_2Z_1)\,\left(Y_1Y_2 - 3\,(X_1 + X_2Z_1 - b\,Z_1)\right) \\
&\quad + (X_1Y_2 + X_2Y_1)\,\left(3X_1X_2 - 3\,Z_1\right).
\end{aligned}
$$

This avoids the multiplication operations for the terms involved with Z_2.

Scalar Multiplication

In Elliptic Curve cryptography, the scalar multiplication

$$
[k]P \;=\; \underbrace{P + P + \cdots + P}_{k\ \text{times}}.
$$

is the fundamental building block for popular cryptographic schemes, such as ECDSA and ECDH. It is crucial to implement the scalar multiplication in constant time to prevent side-channel attacks, while ensuring the performance is efficient even it is done in constant time.

Typically, the scalar value k can be represented as an n-bit value:

$$
k = \sum_{i=0}^{n-1} b_i{\cdot}2^i
$$

where b_i is the i-th bit of the scalar value k.

The scalar multiplication $[k]P$ can be computed using the constant-time Double and Add algorithm, in which the next increment is always doubled ($2{\cdot}P$) in each bit iteration i and accumulated by multiplying with the bit value b_i:

$$
k{\cdot}P = \sum_{i=0}^{n-1} b_i{\cdot}2^i{\cdot}P
$$

Instead of relying on this typical Double and Add algorithm, the implementation uses a *windowed method* to speed up the computation while preserving the constant time property.

The **windowed method** precomputes a table of points P_i for $i \in [0, 2^m - 1]$, where $P_i = [d_i]P$ and d_i is the i-th digit in base 2^m.

With the precomputed points, the typical Double and Add algorithm can be generalized as:

$$[k]P = \sum_{i=0}^{\lceil \frac{n}{m} \rceil - 1} 2^{im} P_i.$$

The implementation uses a 16 points table precomputed, which means that $m = 4$ and the scalar multiplication can be computed as:

$$[k]P = \sum_{i=0}^{\lceil \frac{n}{4} \rceil - 1} 16^i P_i = \sum_{i=0}^{\lceil \frac{n}{4} \rceil - 1} 16^i d_i P.$$

With Horner's method, it computes the scalar multiplication from the most significant digit to the least significant digit:

$$k = ((d_{i-1} \cdot 16 + d_{i-2}) \cdot 16 + d_{i-3}) \cdot 16 + \cdots + d_0$$

In the implementation, the scalar value is firstly encoded in a byte array of size n. Because the precomputed points table is 16 points, the scalar value needs to be processed in 4 bits at a time. Thus, it needs to process the byte array like a 4-bit array.

The `pos` tracks the current position of the bit in the byte array B_i where $i \in [0, \lceil \frac{n}{8} \rceil]$. The `pos` starts from the most significant byte, and decrements 4 bit positions after each windowed process:

$$\text{pos} = n - 4 \cdot i$$

With the current position, it determines which index of the byte array:

$$i = \lceil \frac{\text{pos}}{8} \rceil$$

Then it determins the bit offset of the position, which is either 0 or 4, in that byte B_i:

$$\text{offset} = \text{pos} \mod 8$$

Right shift the offset in the byte to obtain the slot value, which is either top or the bottom 4 bits in the byte:

$$(B_i >> \text{offset}) \mod 16$$

The slot value is computed as:

```
let slot = (k[pos >> 3] >> (pos & 7)) & 0xf;
```

To choose the correct point from the table, it compares the slot value, which is $[0, 15]$, with the index of the precomputed table. When the indexes match, $(\text{slot} \oplus i) - 1$ underflows to the maximum value of the field due to wrap around, and its most significant bit will be 1 for `conditional_assign`.

```
for i in 1..16 {
      t.conditional_assign(
         &pc[i],
         Choice::from(((slot as usize ^ i).wrapping_sub(1) >> 8) as u8 & 1),
      );
}
```

ECDSA Overview

ECDSA has a number of standards, such as FIPS 186-5 (https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf) and SEC1 (https://www.secg.org/sec1-v2.pdf), which specify the algorithm and its parameters.

RustCrypto's ecdsa crate follows the RFC 6979 for deterministic nonce generation instead of relying on random nonce generation, which can be vulnerable to attacks if the same nonce is reused for different messages.

It also handles DER-encoded signatures and OIDs for compatibility with certificate systems, as specified in RFCs 3279, 5758, 5480, 5280 and 5912.

Besides, the crate supports signature normalization to avoid signature malleability attacks, and supports recovering public keys from signatures.

Note that the crate partially implements the hash truncation `bit2int` specified in RFC6979. Even follows the truncation from the standards, it doesn't guarantee resistance to hash collisions. Below explains the implementation details and the security implications.

Signing Algorithm

At the core, the crate uses the following ECDSA signing algorithm:

1. Hash the Message

Compute the cryptographic hash of the message m:

$$e = \mathrm{HASH}(m).$$

Let $L_n = \lceil \log_2 n \rceil$ be the bit-length of the curve order n, and define

$$z = \text{the leftmost } L_n \text{ bits of } e,$$

and reduce z via modulo n

$$z_2 = z \; mod \; n$$

2. Generate Nonce k

Deterministically generate k based on z_2, private key d and an optional entropy via HMAC, such that

$$0 < k < n.$$

3. Compute Point R

Multiply the generator point G by k:

$$R = k \times G = (x_1, \, y_1).$$

4. Compute r

Reduce the x-coordinate modulo n:

$$r = x_1 \bmod n$$

5. Compute s

Compute the signature's second component:

$$s = k^{-1}\left(z_2 + r\,d\right) \bmod n,$$

where d is your private key.

6. Output the Signature

The final signature is the pair (r, s).

Hazmat Functions

The hazmat functions in the RustCrypto `ecdsa` crate are low-level functions that provide the fundamental algorithms for the crate's higher-level ECDSA functions. These hazmat functions require careful handling due to their security assumptions. These functions include:

- **bits2field**: Converts a hash value e into field bytes to represent z. It handles truncation and padding to ensure the hash is within a certain range on a specific curve.
- **sign_prehashed**: Signs a pre-hashed message z_2 using the private key d and nonce k. It is up to the caller to ensure that the nonce k is not reused.
- **sign_prehashed_rfc6979**: Uses RFC 6979's deterministic nonce generation, ensuring that the generated nonce k is unique for each combination of truncated message z and private key d.
- **verify_prehashed**: Verifies a pre-hashed signature (r, s) against the public key dG and message hash z.

Hash Truncation

At step 1, the hash value e is truncated to the leftmost L_n bits to z.

The implementation `bits2field` partially follows the `bit2int` defined in [RFC6979 § 2.3.2]. The `bits2field` function requires the hash value to be at least half the field size (e.g., 128 bits for the P-256 curve). This is an additional security check, deviated from the standards, which do not mandate a minimum hash size.

```
pub fn bits2field<C: EcdsaCurve>(bits: &[u8]) -> Result<FieldBytes<C>> {
    // Minimum allowed bits size is half the field size
    if bits.len() < C::FieldBytesSize::USIZE / 2 {
        return Err(Error::new());
    }

    let mut field_bytes = FieldBytes::<C>::default();

    match bits.len().cmp(&C::FieldBytesSize::USIZE) {
        cmp::Ordering::Equal => field_bytes.copy_from_slice(bits),
        cmp::Ordering::Less => {
            // If bits is smaller than the field size, pad with zeroes on the left
            field_bytes[(C::FieldBytesSize::USIZE - bits.len())..].copy_from_slice(bits);
        }
        cmp::Ordering::Greater => {
            // If bits is larger than the field size, truncate
            field_bytes.copy_from_slice(&bits[..C::FieldBytesSize::USIZE]);
        }
    }

    Ok(field_bytes)
}
```

The truncation ensures that the bit length of z is not greater than the scalar field of the elliptic curve. But this doesn't guarantee that it can entirely avoid the hash collisions due to the hash value overflowing the scalar field, as explained later below.

The L_n is determined by the `FieldBytesSize` defined for a curve. For P-256, its field size in bits happens to be multiple of 8, so the `FieldBytesSize` is 32 bytes.

```
impl elliptic_curve::Curve for NistP256 {
    /// 32-byte serialized field elements.
    type FieldBytesSize = U32;

    TRUNCATED...
}
```

Even though the truncated hash value z is having the same bit length as the scalar modulus n, it is not guaranteed that the hash value is in the range of $[0, n)$. For example, the z is in the range $[0, 2^{256})$, but the upper bound scalar modulus n is far less than 2^256. So the z may overflow the field size.

This overflow issue can be demonstrated in the following test:

```
#[test]
fn rfc6979() {
    let x = hex!("c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721");
    let signer = SigningKey::from_bytes(&x.into()).unwrap();
    let digest1 = hex!
("ffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632552");
    let digest2 = hex!
("0000000000000000000000000000000000000000000000000000000000000001");
    let signature1: Signature = signer.sign_prehash(&digest1).unwrap();
    let signature2: Signature = signer.sign_prehash(&digest2).unwrap();

    assert_eq!(
        signature1.to_bytes().as_slice(),
        signature2.to_bytes().as_slice()
    );
}
```

For P-256, the set of 256-bit integers $\geq n$ has size roughly

$$2^{224} \quad (\text{since } n \approx 2^{256} - 2^{224} + \cdots),$$

so about one in 2^{32} random SHA-256 outputs will lie above n and "wrap around". Collisions *do* exist, but for a uniformly random 256-bit hash they're vanishingly rare in practice.

Therefore, the truncation specified in the standards, such as RFC6979, is to reduce the chances of hash collisions rather than entirely eliminate them.

The security implication also depends on how the verifier incorporates the hash value z. If the application directly uses the hash value supplied by the prover instead of generating the hash, then the prover can take advantage of this overflow issue to do replay attacks. For example, the prover can reuse a signature (r, s) but with a different hash value if the application relies on the hash value for identity purposes, such as a key for nullifier.

Furthermore, note that for the curve P-521, the field size is not a multiple of 8, so the `FieldBytesSize` is 66 bytes, which is equivalent to 528 bits, leading to 7 more bits (528 - 521) for hash collisions. Beside the higher

chances of hash collisions, it doesn't follow the RFC6979 standard, which requires the hash value to be at least the same size as the field size.

```
impl elliptic_curve::Curve for NistP521 {
    /// 66-byte serialized field elements.
    type FieldBytesSize = U66;

    TRUNCATED...
}
```

Deterministic Nonce Generation (RFC 6979)

One of the potential problems of directly using the hazmat function `sign_prehashed` is the risk of nonce reuse, which will lead to private key recovery. The nonce k must be unique for each signature.

The hazmat function `sign_prehashed_rfc6979` implements the deterministic nonce method specified in the RFC 6979:

```
pub fn sign_prehashed_rfc6979<C, D>(
    d: &NonZeroScalar<C>,
    z: &FieldBytes<C>,
    ad: &[u8],
) -> Result<(Signature<C>, RecoveryId)>
where
    C: EcdsaCurve + CurveArithmetic,
    D: Digest + BlockSizeUser + FixedOutput + FixedOutputReset,
    SignatureSize<C>: ArraySize,
{
    let z2 = <Scalar<C> as Reduce<C::Uint>>::reduce_bytes(z);
    let k = NonZeroScalar::<C>::from_repr(rfc6979::generate_k::<D, _>(
        &d.to_repr(),
        &C::ORDER.encode_field_bytes(),
        &z2.to_repr(),
        ad,
    ))
    .unwrap();
    sign_prehashed(d, &k, z)
}
```

The nonce k is generated using HMAC, seeded with the private key d, curve order n, reduced hash value z_2, and optional additional data ad (extra entropy or domain separation). This ensures k is unique per message and private key, eliminating reuse risks.

Recovery ID

From the `sign_prehashed`, along with the signature, a recovery ID is also returned. The recovery ID is used to recover the public key from the signature and message. It is particularly useful for checking if a signature is corresponding to a specific public key.

Recall that in the public key recovery algorithm, the potential public key can be calculated as:

$$R = (x_1, y_1)$$

where x_1 can be the residual of r modulo n, and y_1 can be calculated by plugging x_1 into the elliptic curve equation $y^2 = x^3 + ax + b$, in which a and b defined for specific curve.

Because the point is a field element on the field F_p, which can be greater than curve order n, so the x_1 can be either r or $r + n$. Meanwhile, the y_1 can be either y_1 or $-y_1$.

To obtain the correct public key, the recovery Id indicates which of the possible R points corresponds to the signature:

```
let recovery_id = RecoveryId::new(R.y_is_odd().into(), x_is_reduced);
```

Normalize Signature

ECDSA signatures are malleable because signatures (r, s) and $(r, -s)$ are both valid for the same message. To prevent the verifier from accepting both signatures, the `sign_prehashed` function normalizes the s value to ensure it is in the lower half of the range. This is done by checking if s is greater than half the curve order n and adjusting it accordingly.

The crate provides an interface `NORMALIZE_S` for the curve implementation to define whether `sign_prehashed` should normalize the s value.

```
if C::NORMALIZE_S {
    recovery_id.0 ^= s.is_high().unwrap_u8();
    signature = signature.normalize_s();
}
```

The `is_high()` determines if $s > n/2$, and `normalize_s()` adjusts s. Since adjusting s to $s' = n - s$ is equivalent to replacing the nonce k by $-k$, the new nonce produces the public key point $R' = (-k)G$, which flips the y-coordinate. The recovery ID needs to be adjusted accordingly to indicate the new point R'.

Signature Format

The signature (r, s) is encoded in a format called DER (Distinguished Encoding Rules), ensuring that the signature can be transmitted and stored in a standardized way.

OID (Object Identifier) is a unique identifier used to specify the algorithm used for signing. In the context of ECDSA, the OID indicates the specific elliptic curve and hash function used in the signature. For example, the OID for ECDSA with SHA-256 with the P-256 curve is `1.2.840.10045.3.1.7`.

In the RustCrypto ecdsa crate, the combination of the signature and the OID can be represented by the struct `SignatureWithOid<C>`.

```
pub struct SignatureWithOid<C: EcdsaCurve> {
    signature: Signature<C>,

    oid: ObjectIdentifier,
}
```

Verification Algorithm

Given a public key Q, message m or pre-hash e, and signature (r, s), the verification algorithm is as follows:

1. Validate the ranges (r, s)

$$0 < r, s < n$$

2. Hash and truncate

If not pre-hashed, compute $e = \mathrm{HASH}(m)$ and set

$$z = \text{leftmost } L_n \text{ bits of } e.$$

3. Inverse

$$w = s^{-1} \pmod{n}.$$

4. Twin scalars

$$u_1 = z\,w \pmod{n}, \qquad u_2 = r\,w \pmod{n}.$$

5. Joint multiplication

$$(x_1, y_1) = u_1 G + u_2 Q.$$

Reject the point at infinity.

6. Final check

Accept only if

$$r \equiv x_1 \pmod{n}.$$

Whether to reject signatures where $s > n/2$ to prevent malleability, it is up to the `NORMALIZE_S` definition of the curve or how the caller uses the verify API.

Verification Implementations

The verification can be done either by hashing the message and then verifying the signature, or by providing hash and verifying the signature directly.

With `SignatureWithOid` to represent the signature and the hashing algorithm, it hashes the message with corresponding hashing algorithm to different OID before verifying the signature:

```
impl<C> Verifier<SignatureWithOid<C>> for VerifyingKey<C>
where
   C: EcdsaCurve + CurveArithmetic + DigestPrimitive,
   SignatureSize<C>: ArraySize,
{
   fn verify(&self, msg: &[u8], sig: &SignatureWithOid<C>) -> Result<()> {
       match sig.oid() {
           ECDSA_SHA224_OID => self.verify_prehash(&Sha224::digest(msg),
sig.signature()),
           ECDSA_SHA256_OID => self.verify_prehash(&Sha256::digest(msg),
sig.signature()),
           ECDSA_SHA384_OID => self.verify_prehash(&Sha384::digest(msg),
sig.signature()),
           ECDSA_SHA512_OID => self.verify_prehash(&Sha512::digest(msg),
sig.signature()),
           _ => Err(Error::new()),
       }
   }
}
```

Otherwise, it verifies with pre-hashed message:

```
impl<C> PrehashVerifier<Signature<C>> for VerifyingKey<C>
where
    C: EcdsaCurve + CurveArithmetic,
    SignatureSize<C>: ArraySize,
{
    fn verify_prehash(&self, prehash: &[u8], signature: &Signature<C>) -> Result<()> {
        if C::NORMALIZE_S && signature.s().is_high().into() {
            return Err(Error::new());
        }

        hazmat::verify_prehashed::<C>(
            &ProjectivePoint::<C>::from(*self.inner.as_affine()),
            &bits2field::<C>(prehash)?,
            signature,
        )
    }
}
```

Performance Optimizations

The arithmetic operations involved in ECDSA signing and verification can be categorized into two types: scalar multiplication and field element operations.

The point additions and doubling, which are performed in the calculations like public key dG and $u_1G + u_2Q$, can be optimized through the Montgomery reduction. This is because in projective coordinates, it needs to deal with more multiplications. With the Montgomery reduction, the overhead of repeated multiplications can be amortized.

For the scalar field operations, such as $s = k^{-1}(z + rd)$, can be optimized by Barrett reduction.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	p256/src/arithmetic/field.rs	`try_from_rng` For Base Field in p256 Can Get Invalid Element	**Medium**
#01	p256/src/arithmetic/field.rs	Mixing `FieldElement` and `U256` Types in p256 Base Field Can Cause Errors	**Low**

#00 - `try_from_rng` For Base Field in p256 Can Get Invalid Element

Severity: Medium **Location:** p256/src/arithmetic/field.rs

Description. The base field in p256 is represented in the Montgomery form. The `try_from_rng` function samples a 512-bit value and performs Montgomery reduction to get a field element in the canonical form.

```
fn try_from_rng<R: TryRngCore + ?Sized>(rng: &mut R) -> Result<Self, R::Error> {
    // We reduce a random 512-bit value into a 256-bit field, which results in a
    // negligible bias from the uniform distribution.
    let mut buf = [0; 64];
    rng.try_fill_bytes(&mut buf)?;
    let buf = U512::from_be_slice(&buf);
    Ok(Self(field_impl::from_bytes_wide(buf)))
}

pub(super) fn from_bytes_wide(a: U512) -> U256 {
    let words = a.to_limbs();
    montgomery_reduce(
        U256::new([words[4], words[5], words[6], words[7]]),
        U256::new([words[0], words[1], words[2], words[3]]),
    )
}
```

The Montgomery reduction can reduce $\overline{x}R^{-1}$ to the canonical form (i.e., in the range of `[0, m)`), which requires that the input $\overline{x}$ is no greater than mR. However, the random 512-bit value can be greater than mR. In this case, the `montgomery_reduce` function may not successfully reduce the xR^{-1} into the range `[0, m)`. Instead, the result may be greater than m.

The test below demonstrates such a case. It performs Montgomery reduction on a 512-bit integer and gets an element that is greater than the field modulus.

```
// add this test to p256/src/arithmetic/field.rs
#[test]
fn test_montgomery_wide_word() {
    let t = U512::new([Limb::MAX, Limb::MAX, Limb::MAX, Limb::MAX, Limb::MAX,
        Limb::from_u64(u64::MAX - 0xFFFFFFFF00000000), Limb::ZERO,
Limb::from_u64(u64::MAX - 0x00000000FFFFFFFE)]);
    let r = FieldElement(field_impl::from_bytes_wide(t));
    assert!(r.0 < MODULUS.0); // this will fail
}
```

The `FieldElement` assumes that its inner value is always in the canonical form. Otherwise some operations on the field may get incorrect results. In the test below, `r` and `r_canonical` are the same element in the field but the `neg` operation will get different results. This shows that the operations on "non-canonical" field elements may produce incorrect results.

```
// add this test to p256/src/arithmetic/field.rs
#[test]
fn test_montgomery_incorrect_result() {
    let t = U512::new([Limb::MAX, Limb::MAX, Limb::MAX, Limb::MAX,
        Limb::MAX, Limb::from_u64(u64::MAX - 0xFFFFFFFF00000000), Limb::ZERO,
Limb::from_u64(u64::MAX - 0x00000000FFFFFFFE)]);
    let r = FieldElement(field_impl::from_bytes_wide(t));

    // r.0 is in the range of (m, 2*m)
    assert!(r.0 > MODULUS.0);
    assert!(r.0 - MODULUS.0 < MODULUS.0);

    // get the canonical representation
    let r_canonical = FieldElement(r.0 - MODULUS.0);
    // r and r_canonical are the same element in the field
    assert_eq!(r.to_canonical(), r_canonical.to_canonical());

    // this will fail
    assert_eq!(r.neg(), r_canonical.neg());
}
```

The p256 ecdsa does not use the `try_from_rng` function so it will not affect the signature verification. However, it might cause issues for those external applications that depend on the function.

Recommendation. It's suggested to avoid performing Montgomery reduction on the full 512-bit value. We notice that the `try_from_rng` function in other curve implementations use the reject sampling method to generate a canonical form field element. The function in p256 can adopt a similar approach.

```
fn try_from_rng<R: TryRngCore + ?Sized>(rng: &mut R) -> core::result::Result<Self,
R::Error> {
    let mut bytes = <FieldBytes>::default();

    loop {
        rng.try_fill_bytes(&mut bytes)?;
        if let Some(fe) = Self::from_bytes(&bytes).into() {
            return Ok(fe);
        }
    }
}
```

#01 - Mixing `FieldElement` and `U256` Types in p256 Base Field Can Cause Errors

Severity: Low **Location:** p256/src/arithmetic/field.rs

Description. The `to_canonical` function in the p256 base field is responsible for converting an element from Montgomery form (xR^{-1}) back to its plain form (x). The current implementation is as follows:

```
/// Translate a field element out of the Montgomery domain.
#[inline]
pub(crate) const fn to_canonical(self) -> Self {
    Self(field_impl::to_canonical(self.0))
}
```

While the function aims to return the canonical form of the field element, it introduces potential issues. Specifically, the function returns a `FieldElement` type, which implicitly assumes that the element is still in Montgomery form. This could confuse users because `FieldElement` elements are expected to be in Montgomery form at all times. Returning a non-Montgomery element as a `FieldElement` might lead to unexpected behavior, particularly when `to_canonical` is called multiple times in succession.

Additionally, the `MODULUS` is defined as a `FieldElement` type:

```
/// Constant representing the modulus
/// p = 2^{224}(2^{32} - 1) + 2^{192} + 2^{96} - 1
pub const MODULUS: FieldElement = FieldElement(U256::from_be_hex(MODULUS_HEX));
```

This could further contribute to confusion, as the `MODULUS` is not in Montgomery form but is represented as a `FieldElement`.

Recommendation. It is recommended to change the return type of the `to_canonical` function to `U256`. This type represents the raw, plain field element and avoids the confusion of returning it wrapped in a `FieldElement`. Similarly, consider redefining the `MODULUS` constant to use the `U256` type directly, ensuring consistency and clarity in the representation of raw field elements.