Entropy/Rust Cryptography and Implementation Review

Entropy Cryptography

Version 1.0 – Final Report – August 25, 2023

Prepared By
Eric Schorn
Kevin Henry
Javed Samuel

Prepared For
Tux Pacific
Bogdan Opanchuk

1 Executive Summary

Synopsis

During the summer of 2023, Entropy Cryptography Inc. engaged NCC Group's Cryptography Services team to perform a cryptography and implementation review of several Rust-based libraries implementing constant-time big integer arithmetic, prime generation, and secp256k1 (k256) elliptic curve functionality. Two consultants performed the review over 40 person-days (including retesting) with support provided over a private Discord channel.

The initial review resulted in 2 high severity, 7 low severity, and 3 informational findings across the in-scope repositories. These findings were promptly addressed by Entropy and retested as part of this engagement. Two low-severity findings remain as 'Risk Accepted' as they are dependent on external requirements.

Overall, the reviewed code appeared to be well-architected, robustly implemented, and aligned to specifications. Documentation was found to be of high quality, with helpful comments, annotations, and references where appropriate. Identified issues were addressed promptly once reported, with the proposed fixes aligning with the recommendations made in this report.

Scope

The three primary code repositories in scope for this review were:

1. Commits `fde0661` and `ad08a7e` of *github.com/RustCrypto/crypto-bigint*.
2. Commits `c6bbdf3` and `7d95cf6` of *github.com/entropyxyz/crypto-primes*.
3. Commits `0f27814` and `5c829a4` of *github.com/RustCrypto/elliptic-curves/k256*.

A variety of supporting resources were used including a technical paper, the associated test cases, and Rust documentation. The testing methodology primarily relied upon manual source code inspection and analysis, comparison to specifications and reference materials, and detailed timing measurements made on a Cortex-M4 based MCU system.

Limitations

No significant obstacles were encountered during the project and robust coverage of the target code was successfully achieved. The direct dependencies received light review, primarily around API usage.

Key Findings

The initial review uncovered a variety of issues, including:

- **Missing Validation of 'low s' ECDSA Signatures** that may introduce application incompatibilities stemming from inherently malleable ECDSA signatures.
- **Missing Schnorr Signature Verification Check** which allows two valid signatures to be verified for the same message and public key, effectively violating BIP 340's objective of signature non-malleability.
- **Timing Variability in ECDSA Signature Generation** where a benign timing variability may complicate the detection of other timing side-channels that leak secret values.
- **Inexact Secret Key Deserialization** that introduces validation complexity along with potential issues involving malleability and interoperability.
- **Square Root Computation is not Constant Time** which may lead to the leakage of secret material.

The project concluded with a retest phase that confirmed all findings except two were fixed. These remaining low-severity findings were marked 'Risk Accepted' due to dependence on external requirements.

Strategic Recommendations

NCC Group recommends the prioritization of several emergent themes for future development efforts, including:

1. Continue to maintain the current high quality of documentation as new features and functionality are added over time.
2. Develop additional detailed documentation for the *crypto-bigint* functionality that highlights supported use cases, particularly related to modular arithmetic involving both compile-time and run-time moduli.
3. Develop and enable CI/CD test coverage reporting on each repository.
4. Implement and document a prime-generation mode compliant with FIPS 186-5.

2 Dashboard

Target Data		Engagement Data	
Name	Crypto bigInt, crypto primes and k256 curve	**Type**	Cryptography and implementation review
Type	Source code libraries	**Method**	Source code analysis, MCU timing measurements
Platforms	Rust	**Dates**	2023-06-07 to 2023-08-11
Environment	Specific repository commits	**Consultants**	2
		Level of Effort	40 person-days

Targets

github.com/RustCrypto/crypto-bigint	Big integer support for cryptographic applications
github.com/entropyxyz/crypto-primes	Prime number generation for cryptographic applications
github.com/RustCrypto/elliptic-curves/k256	Secp256k1 elliptic curve support

Finding Breakdown

Critical issues	0	
High issues	2	🟥🟥
Medium issues	0	
Low issues	7	🟨🟨🟨🟨🟨🟨🟨
Informational issues	3	⬜⬜⬜
Total issues	**12**	

Category Breakdown

Cryptography	3	🟥🟥🟨
Data Exposure	4	🟨🟨🟨🟨
Data Validation	2	🟨🟨
Error Reporting	1	⬜
Other	1	⬜
Patching	1	⬜

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Missing Validation of 'low s' ECDSA Signatures	Fixed	2VF	High
Missing Schnorr Signature Verification Check	Fixed	CRR	High
Missing Minimum `max_bit_length` Check	Fixed	K2E	Low
Hex Decoding for `Uint` is not Constant Time	Fixed	VVV	Low
Minor Timing Leak in Saturating Arithmetic Operations	Fixed	HRB	Low
Square Root Computation is not Constant Time	Fixed	K34	Low
Inexact Secret Key Deserialization	Risk Accepted	FQT	Low
Minor Timing Leaks in Wide Scalar Arithmetic	Fixed	UNR	Low
Timing Variability in ECDSA Signature Generation	Risk Accepted	ENP	Low
Missing Toolchain Specification and Outdated Dependencies	Fixed	YTT	Info
Silent Overflow/Wrapping Condition	Fixed	QTR	Info
`random_mod` Tests Expect Incorrect Behavior	Fixed	NEW	Info

4 Finding Details

High Missing Validation of 'low s' ECDSA Signatures

Overall Risk	High	**Finding ID**	NCC-E008526-2VF
Impact	Medium	**Component**	k256
Exploitability	High	**Category**	Cryptography
		Status	Fixed

Impact

An API (and implemented functionality) that does not fully handle 'low s' signature verification constraints may introduce application incompatibilities stemming from inherently malleable ECDSA signatures.

Description

ECDSA signatures consist of two scalars `r` and `s` (both `mod n`, where `n` is the order of the underlying elliptic curve). Each valid signature `r, s` has a counterpart `r, -s mod n` that is also valid. The second signature can be derived from the first by any party without knowing the secret key (and only on the same message). This malleability has caused issues with consensus systems such as Bitcoin[1] in the past. As a result, some systems restrict the range of `s` to be the smaller of `s` and `-s mod n` during signature generation and also validate this constraint during signature verification[2].

The *k256* signature generation process utilizes the `normalize_s()` helper function from within the *ecdsa.rs* source file[3] to perform this range adjustment. This code is partially excerpted in finding "Timing Variability in ECDSA Signature Generation". As a result, all signatures produced are of the restricted 'low s' form.

However, the *k256* signature verification process accepts both forms of signature and does not have a simple way of validating the range of `s` if desired. This was confirmed by a short fuzz-type test where the `normalize_s()` function was eliminated during signature generation without any issues encountered in signature verification. Accepting the broader range of `s` contradicts the *k256 README.md* documentation[4], as it states:

> Support for ECDSA/secp256k1 signing and verification, applying low-s normalization (BIP 0062) as used in consensus-critical applications, and additionally supports secp256k1 public-key recovery from ECDSA signatures (as used by e.g. Ethereum)

This situation likely stems from the specialized requirements of Bitcoin on k256 and their non-applicability to the other (predecessor) curves. While the user may be able to perform this check outside of the library, the documentation suggests they would not need to, they may not know how, and they may be time pressured. Additionally, some applications may not want to enforce this constraint on signature verification.

1. https://eklitzke.org/bitcoin-transaction-malleability
2. https://github.com/bitcoin-core/secp256k1/blob/2bd5f3e6184f1a4cfaed910ab74269fb6ab635be/src/secp256k1.c#L455
3. https://github.com/RustCrypto/elliptic-curves/blob/0f27814b47f4ea2a43f2958e91d142688c89d89a/k256/src/ecdsa.rs#L224
4. Item 2 of https://github.com/RustCrypto/elliptic-curves/tree/0f27814b47f4ea2a43f2958e91d142688c89d89a/k256#supported-algorithms

As a result, systems built with the *k256* implementation will always produce low-s signatures but may unintentionally (and silently) accept both varieties. The latter case may introduce signature malleability incompatibilities, particularly in consensus-oriented systems where messages may be identified by the signature byte string.

Recommendation

Adapt the API to incorporate validation of the low-s constraint. Alternatives include: A) validate the low-s form in all cases, B) add a boolean flag to enable validation of the low-s form, C) add a separate API that includes low-s validation.

Location

The `verifying_key.verify()` API ultimately resolves to the `verify_prehashed()` function implemented in *RustCrypto/signatures/ecdsa/src/hazmat.rs*.

Retest Results

2023-07-31 – Fixed

NCC Group reviewed changes to *k256/src/ecdsa.rs* present in commit `5c829a4` and observed a new validation check that throws an error when `sig.s().is_high().into()`. This is aligned with the recommendation. As such, this finding has been marked ‘Fixed’.

High

Missing Schnorr Signature Verification Check

Overall Risk	High	**Finding ID**	NCC-E008526-CRR
Impact	High	**Component**	k256
Exploitability	High	**Category**	Cryptography
		Status	Fixed

Impact

A missing validation check allows two valid signatures to be verified for the same message and public key, effectively violating BIP 340's objective of signature non-malleability.

Description

The Schnorr signature verification algorithm defined in BIP 340 is shown below. Note the three checks implemented as steps 7, 8 and 9.

```
1. The algorithm Verify(pk, m, sig) is defined as:
2. Let P = lift_x(int(pk)); fail if that fails.
3. Let r = int(sig[0:32]); fail if r ≥ p.
4. Let s = int(sig[32:64]); fail if s ≥ n.
5. Let e = int(hashBIP0340/challenge(bytes(r) || bytes(P) || m)) mod n.
6. Let R = s·G - e·P.
7. Fail if is_infinite(R).
8. Fail if not has_even_y(R).
9. Fail if x(R) ≠ r.
10. Return success iff no failure occurred before reaching this point.
```

The k256 Schnorr `verify_prehash()` function implemented in the *verifying.rs* source file is shown below. Lines 69-75 of the implementation align with step 5 above, lines 77-83 align with step 6, and lines 85-87 align with steps 8 and 9. While the `to_affine()` function on line 83 is able to return the `AffinePoint::IDENTITY` value, there is no check corresponding to step 7 above.

```
60 impl PrehashVerifier<Signature> for VerifyingKey {
61     fn verify_prehash(
62         &self,
63         prehash: &[u8],
64         signature: &Signature,
65     ) -> core::result::Result<(), Error> {
66         let prehash: [u8; 32] = prehash.try_into().map_err(|_| Error::new())?;
67         let (r, s) = signature.split();
68
69         let e = <Scalar as Reduce<U256>>::reduce_bytes(
70            &tagged_hash(CHALLENGE_TAG)
71                .chain_update(signature.r.to_bytes())
72                .chain_update(self.to_bytes())
73                .chain_update(prehash)
74                .finalize(),
75         );
76
77         let R = ProjectivePoint::lincomb(
78            &ProjectivePoint::GENERATOR,
79            s,
80            &self.inner.to_projective(),
81            &-e,
82          )
83          .to_affine();
84
85          if R.y.normalize().is_odd().into() || R.x.normalize() != *r {
86              return Err(Error::new());
87          }
88
89          Ok(())
90      }
91  }
```

As a result, two signatures for the same message and public key will pass verification. The first signature is a correctly generated (r_1, s_1) pair. The second (r_2, s_2) pair is generated as follows:

1. Set r_2 to `0`.
2. Recalculate e by performing step 5.
3. Set $s_2 = e \cdot sk$
4. Given P = sk·G, note that step 6 now becomes R = e·sk·G - e·sk·G
   R will be the point at infinity; internally, this is `AffinePoint { x=0, y=0, infinity=1 }`.
5. Since step 7 is not present, steps 8 and 9 will both pass and step 10 will incorrectly return success.

Thus the second signature is (0, e · sk). Both signatures will verify for the same message m and public key P. Admittedly, the adversary has effectively disclosed their sk (as it can be easily derived from s_2). This disclosure may be worthwhile if it causes the victim to make a high-value decision incorrectly.

As an informational aside, the validation step 4 above allows for s=0 but the code puts this into a `NonZeroScalar` (which disallows 0) resulting in a minor mismatch between algorithm and implementation. Note that the deserialization of `r` as a `FieldElement` does indeed correctly allow a value of 0. Additionally, the deserialization process correctly checks the upper limits of step 4 and 5.

Recommendation

Implement a check for the `AffinePoint::IDENTITY` corresponding to step 7 just prior to (or on) line 85.

Location

RustCrypto/elliptic-curves/k256/src/schnorr/verifying.rs

Retest Results

2023-08-01 – Fixed

NCC Group reviewed changes to *k256/src/schnorr/verifying.rs* present in commit `9e905f0` and observed a new validation check involving `R.is_identity().into()`. This is aligned with the recommendation. As such, this finding has been marked 'Fixed'.

Low

Missing Minimum `max_bit_length` Check

Overall Risk	Low	**Finding ID**	NCC-E008526-K2E
Impact	Low	**Component**	crypto-primes
Exploitability	High	**Category**	Data Validation
		Status	Fixed

Impact

When the `hazmat::Sieve::new()` function is called with a `max_bit_length()` of `0usize`, the user may encounter an overflow, uncontrolled panic or compilation issue.

Description

The publicly visible `hazmat::random_odd_uint()` function implemented in *sieve.rs* checks that its target bit length parameter is nonzero. However, the sibling and publicly visible `hazmat::Sieve::new()` function does not have this check. If the latter function were supplied with an argument of `0usize`, then subtracting `1` on line 92 would cause an overflow, panic or compilation issue (depending upon context).

```
81  pub fn new(start: &Uint<L>, max_bit_length: usize, safe_primes: bool) -> Self {
82      if max_bit_length > Uint::<L>::BITS {
83          panic!(
84              "The requested bit length ({}) is larger than the chosen Uint size",
85              max_bit_length
86          );
87      }
88
89      // If we are targeting safe primes, iterate over the corresponding
90      // possible Germain primes (`n/2`), reducing the task to that with `safe_primes =
        ↪ false`.
91      let (max_bit_length, base) = if safe_primes {
92          (max_bit_length - 1, start >> 1)
93      } else {
94          (max_bit_length, *start)
95      };
96  ...
```

Note that logic subsequent to that shown above handles a number of other corner cases. It appears that only this condition can cause an issue prior to the handling logic.

Recommendation

In the `hazmat::Sieve::new()` function, validate that the supplied `max_bit_length` is non-zero (near line 87).

Location

crypto-primes/src/hazmat/sieve.rs

Retest Results

2023-07-31 – Fixed

NCC Group reviewed changes to *crypto-primes/src/hazmat/sieve.rs* present in commit `7d95cf6` and observed a new validation check that panics when `max_bit_length == 0`. This is aligned with the recommendation. As such, this finding has been marked ‘Fixed’.

Low

Hex Decoding for `Uint` is not Constant Time

Overall Risk	Low	**Finding ID**	NCC-E008526-VVV
Impact	Medium	**Component**	crypto-bigint
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Decoding of hex bytes is not constant time and the associated functions are not annotated as `vartime` within the library, which may lead to unsafe usage by other applications.

Description

The *crypto_bigint* library provides the function `from_be_hex()` to construct a `Uint` from a hex string. Hex characters are decoded byte-by-byte via the function `decode_hex_byte()` in *src/uint/encoding.rs*:

```rust
/// Decode a single byte encoded as two hexadecimal characters.
const fn decode_hex_byte(bytes: [u8; 2]) -> u8 {
    let mut i = 0;
    let mut result = 0u8;

    while i < 2 {
        result <<= 4;
        result |= match bytes[i] {
            b @ b'0'..=b'9' => b - b'0',
            b @ b'a'..=b'f' => 10 + b - b'a',
            b @ b'A'..=b'F' => 10 + b - b'A',
            b => {
                assert!(
                    matches!(b, b'0'..=b'9' | b'a' ..= b'f' | b'A'..=b'F'),
                    "invalid hex byte"
                );
                0
            }
        };

        i += 1;
    }

    result
}
```

The `match` expression represents a conditional branch on the input and will result in a different number of comparisons based on the values of `bytes[i]`. According to The Rust Reference:

> If the scrutinee expression is a value expression, it is first evaluated into a temporary location, and **the resulting value is sequentially compared to the patterns in the arms until a match is found. The first arm with a matching pattern is chosen as the branch target of the match**, any variables bound by the pattern are assigned to local variables in the arm's block, and control enters the block.[5]

5. https://doc.rust-lang.org/reference/expressions/match-expr.html

It was also observed that the `u8` arithmetic in each branch could be different if the compiler were unable to optimize the constant portions of `10 + b - b'a'` versus `b - b'0'`.

The library should make it clear that hex decoding operations are not explicitly constant time or should adopt a different approach to achieve constant-time decoding. It is likely that the variance in timing will not enable a meaningful attack, but it is nevertheless recommended to err on the side of caution. The `rust-ct-codecs` crate appears to provide an alternative approach to hex decoding with stronger timing guarantees[6]. This library was not reviewed as part of this engagement, and is being linked for reference purposes only. Similarly, the Constant-Time Toolkit (CTTK)[7] provides reference implementations for hex decoding that may be readily translated from C to Rust.

Recommendation

Ensure that hex byte decoding does not rely on conditional branches, or annotate the function as `vartime` and ensure this is clearly communicated to users of the library.

Location

src/uint/encoding.rs

Retest Results

2023-08-02 – Fixed

As part of PR #254, commit `73e82bd`, the non-CT `match` statement was replaced in favor of the new constant-time approach implemented with the helper function `decode_nibble()`:

```
const fn decode_nibble(src: u8) -> u16 {
    let byte = src as i16;
    let mut ret: i16 = -1;

    // 0-9  0x30-0x39
    // if (byte > 0x2f && byte < 0x3a) ret += byte - 0x30 + 1; // -47
    ret += (((0x2fi16 - byte) & (byte - 0x3a)) >> 8) & (byte - 47);
    // A-F  0x41-0x46
    // if (byte > 0x40 && byte < 0x47) ret += byte - 0x41 + 10 + 1; // -54
    ret += (((0x40i16 - byte) & (byte - 0x47)) >> 8) & (byte - 54);
    // a-f  0x61-0x66
    // if (byte > 0x60 && byte < 0x67) ret += byte - 0x61 + 10 + 1; // -86
    ret += (((0x60i16 - byte) & (byte - 0x67)) >> 8) & (byte - 86);

    ret as u16
}
```

The revised approach no longer depends on the value of each byte, and therefore addresses the issue; as such, this finding has been marked 'Fixed'.

6. https://github.com/jedisct1/rust-ct-codecs/tree/master
7. https://github.com/pornin/CTTK

Low

Minor Timing Leak in Saturating Arithmetic Operations

Overall Risk	Low	**Finding ID**	NCC-E008526-HRB
Impact	Medium	**Component**	crypto-bigint
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

A conditional branch on the result of an arithmetic operation may introduce a subtle timing attack based on the result of the operation.

Description

The *crypto-bigint* library provides saturating arithmetic operations, where integer overflow is addressed by returning the maximum or minimum value for the underlying type, rather than silently wrapping or returning an error. The following code pattern is used across the implementations for `saturating_add`, `saturating_sub`, and `saturating_mul`; see *uint/add.rs*, for example:

```
    /// Perform saturating addition, returning `MAX` on overflow.
    pub const fn saturating_add(&self, rhs: &Self) -> Self {
        let (res, overflow) = self.adc(rhs, Limb::ZERO);

        if overflow.0 == 0 {
            res
        } else {
            Self::MAX
        }
    }
```

Here, the `adc` function performs addition with a tracked "carry" value, and the saturating operation simply checks if a carry/overflow occurred and returns the appropriate value. The underlying implementation of `==` (`PartialEq`) for `Limb` provides a constant-time comparison. However, the highlighted conditional will use at least one additional instruction to jump to the `else` case if overflow occurs.

The crate implements `CtChoice` for performing certain constant-time comparison operations, including a constant-time select function in *src/ct_choice.rs*:

```
    /// Return `b` if `self` is truthy, otherwise return `a`.
    pub(crate) const fn select(&self, a: Word, b: Word) -> Word {
        a ^ (self.0 & (a ^ b))
    }
```

The functions `saturating_add`, `saturating_sub`, and `saturating_mul` could be updated to leverage the constant-time select function to eliminate the minor timing difference identified above.

Recommendation

Update the identified saturating arithmetic operations to leverage the constant-time `select` function, or an alternative constant-time approach.

Location

- *uint/add.rs*
- *uint/mul.rs*
- *uint/sub.rs*

Retest Results

2023-08-02 – Fixed

As part of PR #256, commit `cf08c2e`, the `saturating_add()` and `saturating_sub()` functions were updated to use `ct_select()` instead of an `if` statement. The issue with `saturating_mul()` was fixed independently as part of PR #253. As such, this finding has been marked ‘Fixed’.

Low

Square Root Computation is not Constant Time

Overall Risk	Low	**Finding ID**	NCC-E008526-K34
Impact	Medium	**Component**	crypto-bigint
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

The `sqrt` function leverages an iterative approach that is not constant time, thereby leading to a timing leak based on the input value.

Description

The `sqrt` function uses a numerical approach to refine estimates of an integer square root. Each iteration moves closer to the correct result until no closer integer result can be achieved. The initial estimate is based on the maximum number of bits needed to represent the value of the input, *not on the maximum value of the input*. Therefore, the expected runtime of the implementation will be based on the value of `self`. The function is not annotated with `_vartime`, suggesting it is intended to be constant time in its execution.

The complete `sqrt()` function in *uint/sqrt.rs* is as follows:

```rust
    /// Computes √(`self`)
    /// Uses Brent & Zimmermann, Modern Computer Arithmetic, v0.5.9, Algorithm 1.13
    ///
    /// Callers can check if `self` is a square by squaring the result
    pub const fn sqrt(&self) -> Self {
        let max_bits = (self.bits_vartime() + 1) >> 1;
        let cap = Self::ONE.shl_vartime(max_bits);
        let mut guess = cap; // ≥ √(`self`)
        let mut xn = {
            let q = self.wrapping_div(&guess);
            let t = guess.wrapping_add(&q);
            t.shr_vartime(1)
        };

        // If guess increased, the initial guess was low.
        // Repeat until reverse course.
        while Uint::ct_lt(&guess, &xn).is_true_vartime() {
            // Sometimes an increase is too far, especially with large
            // powers, and then takes a long time to walk back.  The upper
            // bound is based on bit size, so saturate on that.
            let le = Limb::ct_le(Limb(xn.bits_vartime() as Word), Limb(max_bits as Word));
            guess = Self::ct_select(&cap, &xn, le);
            xn = {
                let q = self.wrapping_div(&guess);
                let t = guess.wrapping_add(&q);
                t.shr_vartime(1)
            };
        }

        // Repeat while guess decreases.
        while Uint::ct_gt(&guess, &xn).is_true_vartime() &&
        ↳xn.ct_is_nonzero().is_true_vartime() {
            guess = xn;
            xn = {
                let q = self.wrapping_div(&guess);
                let t = guess.wrapping_add(&q);
                t.shr_vartime(1)
            };
        }

        Self::ct_select(&Self::ZERO, &guess, self.ct_is_nonzero())
    }
```

The function was modified to locally track the number of iterations of the two `while` loops while executing the provided `simple()` test, which validates the result for the first 13 perfect squares:

- Input: `4`, total loops: `1`
- Input: `9`, total loops: `1`
- Input: `16`, total loops: `2`
- Input: `25`, total loops: `1`
- Input: `36`, total loops: `1`
- Input: `49`, total loops: `1`
- Input: `64`, total loops: `2`
- Input: `81`, total loops: `2`
- Input: `100`, total loops: `2`
- Input: `121`, total loops: `1`
- Input: `144`, total loops: `1`
- Input: `169`, total loops: `1`

The total number of loop iterations varies based on even small changes to the input value, violating any constant-time guarantees.

No direct usage of `sqrt` or the related functions `wrapping_sqrt` and `checked_sqrt` were observed within the crate (outside of tests), so the immediate impact of the finding is low. However, the library conventions suggest that this function should be marked `_vartime` to ensure it is used safely outside of the crate.

Recommendation

Adopt a constant-time square root computation, or mark the function `_vartime` to clearly communicate its behavior.

Location

uint/sqrt.rs

Retest Results

2023-08-02 – Fixed

As part of PR #256, commit `17c4358`, the various `sqrt` functions are now deprecated, e.g.:

```
    /// See [`Self::sqrt_vartime`].
    #[deprecated(
        since = "0.5.3",
        note = "This functionality will be moved to `sqrt_vartime` in a future release."
    )]
```

This update is consistent with the recommendations and serves to notify users that the function is not constant time.

Low

Inexact Secret Key Deserialization

Overall Risk	Low	**Finding ID**	NCC-E008526-FQT
Impact	Medium	**Component**	k256
Exploitability	Low	**Category**	Data Validation
		Status	Risk Accepted

Impact

Allowing any variability in the deserialization of cryptographic material introduces validation complexity along with potential issues involving malleability and interoperability.

Description

The `SecretKey` type declared in *elliptic-curves/k256/src/lib.rs* utilizes the `from_slice()` deserialization trait functionality implemented in *elliptic-curve/src/secret_key.rs* (note the former path is plural, while the latter is singular and from a traits crate). This function is partially excerpted below and allows for variability in the input slice length.

```
/// Deserialize secret key from an encoded secret scalar passed as a
/// byte slice.
///
/// The slice is expected to be at most `C::FieldBytesSize` bytes in
/// length but may be up to 4-bytes shorter than that, which is handled by
/// zero-padding the value.
pub fn from_slice(slice: &[u8]) -> Result<Self> {
    if slice.len() > C::FieldBytesSize::USIZE {
        return Err(Error);
    }

    /// Maximum number of "missing" bytes to interpret as zeroes.
    const MAX_LEADING_ZEROES: usize = 4;

    let offset = C::FieldBytesSize::USIZE.saturating_sub(slice.len());

    if offset == 0 {
        Self::from_bytes(FieldBytes::<C>::from_slice(slice))
    } else if offset <= MAX_LEADING_ZEROES { ... ... ...
```

As indicated in the code comments above, the function allows somewhat variable input length and performs zero-padding as needed. This increases the complexity of validation performed in the enclosing scope, introduces malleability into any enclosing structures, and may impact interoperability. Past experience with Ed25519[8,9] suggests that variability is to be avoided (albeit references are more pertinent to signatures than secret keys). The related `from_bytes()` deserialization function requires an exact input length.

Recommendation

Require and validate an exact length input prior to starting deserialization. Update the docstrings to indicate the required byte-array sizes for deserializing other structs such as the Schnorr `SigningKey` and `VerifyingKey` (for ease of use).

8. https://github.com/dalek-cryptography/ed25519-dalek/issues/20
9. https://eprint.iacr.org/2020/1244.pdf

Location

- *RustCrypto/elliptic-curves/k256/src/lib.rs*
- *RustCrypto/traits/elliptic-curve/src/secret_key.rs*
- *RustCrypto/elliptic-curves/k256/src/schnorr/signing.rs*
- *RustCrypto/elliptic-curves/k256/src/schnorr/verifying.rs*

Retest Results

2023-07-31 – Not Fixed

NCC Group has reviewed discussion in issue 1330 of *RustCrypto/traits* which indicates intentional support for a legacy corner-case described in issue 769. Thus, this finding was not fixed and is considered 'Risk Accepted'.

Low

Minor Timing Leaks in Wide Scalar Arithmetic

Overall Risk	Low	**Finding ID**	NCC-E008526-UNR
Impact	Low	**Component**	k256
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

The compiler may optimize the `ct_less()` function such that its usage may allow a subtle timing attack based on the result of the arithmetic comparison.

Description

The `ct_less()` function implemented in the *wide64.rs* source file is excerpted below. The first code comment and function name indicate the function is constant time, while the second code comment and actual implementation (correctly) reflect the function not being constant time as it omits `Choice`. Note that an optimization barrier is *not* inserted and the function is to be inlined.

```
/// Constant-time comparison.
#[inline(always)]
fn ct_less(a: u32, b: u32) -> u32 {
    // Do not convert to Choice since it is only used internally,
    // and we don't want loss of performance.
    (a < b) as u32
}
```

The `muladd()` function excerpted below utilizes `ct_less()` in two highlighted locations. The compiler will attempt to inline the `ct_less()` logic and optimize across the function in totality. This may introduce subtle timing differences involving overflows based upon secret data.

```
/// Add a*b to the number defined by (c0,c1,c2). c2 must never overflow.
fn muladd(a: u32, b: u32, c0: u32, c1: u32, c2: u32) -> (u32, u32, u32) {
    let t = (a as u64) * (b as u64);
    let th = (t >> 32) as u32; // at most 0xFFFFFFFFFFFFFFFE
    let tl = t as u32;

    let new_c0 = c0.wrapping_add(tl); // overflow is handled on the next line
    let new_th = th + ct_less(new_c0, tl); // at most 0xFFFFFFFFFFFFFFFF
    let new_c1 = c1.wrapping_add(new_th); // overflow is handled on the next line
    let new_c2 = c2 + ct_less(new_c1, new_th); // never overflows by contract (verified in
    ↳ the next line)
    debug_assert!((new_c1 >= new_th) || (new_c2 != 0));
    (new_c0, new_c1, new_c2)
}
```

This scenario is also present in the `sumadd_fast()` and `muladd_fast()` functions, as well as the same four functions within the *wide32.rs* source file.

Recommendation

A constant-time implementation of the carry propagation logic should be achievable with chained `overflowing_add()`[10] operations. Alternatively, a constant-time `Choice` operation could be inserted into `ct_less()` (and used comprehensively across each of the functions). Address instances across both *wide64.rs* and *wide32.rs*.

Location

- *RustCrypto/elliptic-curves/k256/src/arithmetic/scalar/wide64.rs*
- *RustCrypto/elliptic-curves/k256/src/arithmetic/scalar/wide32.rs*

Retest Results

2023-08-02 – Fixed

NCC Group reviewed changes present in *k256/src/arithmetic/scalar/wide64.rs* and *wide32.rs* for PR #917 and observed that the relevant carry chain propagation is now implemented with `overflowing_add()` and `wrapping_add()` exclusively. This is aligned with the recommendation. As such, this finding has been marked 'Fixed'.

10. https://doc.rust-lang.org/std/primitive.u32.html#method.overflowing_add

Low

Timing Variability in ECDSA Signature Generation

Overall Risk	Low	**Finding ID**	NCC-E008526-ENP
Impact	Medium	**Component**	k256
Exploitability	None	**Category**	Cryptography
		Status	Risk Accepted

Impact

A benign timing variability in the signature generation process may complicate the detection of other timing side-channels that leak secret values.

Description

As the signature generation process involves a secret key, it is generally required to operate in constant time. This prevents the possibility of secret leakage from timing side-channels. While it is common to increase confidence in constant-time operation via statistical means[11] on full-featured desktop/server systems[12], a complementary technique involves running on deterministic (embedded) hardware. The k256 signature generation process exhibits approximately 280 cycles of timing variability when running on a Cortex-M4 MCU.

The `try_sign_prehashed()` function implemented in the *ecdsa.rs* source file makes use of a `normalize_s()` helper function on line 224 as excerpted below.

```
219     ...
220     let signature = Signature::from_scalars(r, s)?;
221     let is_r_odd = R.y.normalize().is_odd();
222     let is_s_high = s.is_high();
223     let is_y_odd = is_r_odd ^ is_s_high;
224     let signature_low = signature.normalize_s().unwrap_or(signature);
225     let recovery_id = RecoveryId::new(is_y_odd.into(), false);
226
227     Ok((signature_low, Some(recovery_id)))
228 }
```

The `normalize_s()` helper function is implemented in the *ecdsa* dependency crate as shown below. This is clearly not a constant-time implementation as the amount of calculation following the conditional test on line 303 is unbalanced.

```
296 /// Normalize signature into "low S" form as described in
297 /// [BIP 0062: Dealing with Malleability][1].
298 ///
299 /// [1]: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki
300 pub fn normalize_s(&self) -> Option<Self> {
301     let s = self.s();
302
303     if s.is_high().into() {
304         let mut result = self.clone();
305         result.s = ScalarPrimitive::from(-s);
306         Some(result)
```

11. https://crates.io/crates/dudect-bencher
12. https://github.com/RustCrypto/crypto-bigint/blob/master/dudect/src/main.rs

```
    } else {
        None
    }
}
```

As the above timing variability does not involve secret data, this finding severity has been rated 'Low'. However, it may obscure the presence and detection of additional timing side-channels that do involve secret data.

Recommendation

The `normalize_s()` function may already be utilized by other applications, which increases the impact of its variability and makes the function signature hard to change. Nonetheless, several approaches are possible, including:

1. Introduce a sibling function alongside `normalize_s()` that returns a conditional selection in constant time[13] rather than an `Option`. Adapt existing callers to utilize this function as possible. Mark the original function as deprecated.
2. As an initial pragmatic step that does not require changes in dependencies, replace the call to `normalize_s()` on line 224 with the following:

   ```
   let signature = Signature::from_scalars(r, s)?;
   let signature2 = Signature::from_scalars(r, -s)?;
   let signature_low = if is_s_high.into() {signature2} else {signature};
   ```

   This was implemented and the signing process confirmed to be constant time on the Cortex-M4. However, this may be a brittle solution for other architectures and compiler versions. Note that the overall signing process takes several million cycles, so the relative overhead is negligible as `from_scalars` does not perform any point operations.
3. It could be argued that non-secret related timing variability is actually a good thing as it makes an adversary's task of secret-related recovery harder. However, the non-secret variability can easily be removed in this instance since it is deterministic based on `s`.

Location

- *elliptic-curves/k256/src/ecdsa.rs*
- *RustCrypto/signatures/blob/ecdsa/v0.16.7/ecdsa/src/lib.rs*

Retest Results

2023-08-08 – Not Fixed

NCC Group has reviewed this finding with the development team and agreed on the proper path forward. As the fix involves API compatibility logistics with an external dependency, this work has been scheduled on the near-term roadmap. Thus, this finding was not fixed at retest time and is considered 'Risk Accepted'.

13. https://docs.rs/subtle/2.5.0/subtle/trait.ConditionallySelectable.html#tymethod.conditional_select

Info

Missing Toolchain Specification and Outdated Dependencies

Overall Risk	Informational	**Finding ID**	NCC-E008526-YTT
Impact	Undetermined	**Component**	crypto-bigint, crypto-primes, k256
Exploitability	Undetermined	**Category**	Patching
		Status	Fixed

Impact

An unspecified minimum toolchain version may allow code to build on an unsupported version with unexpected results. Attackers may attempt to identify and utilize publicly known vulnerabilities in outdated dependencies to exploit the functionality of the target code.

Description

The *crypto-bigint/Cargo.toml* and *elliptic-curves/k256/Cargo.toml* manifest files specify the minimum supported toolchain version via the `[package] rust-version = “1.65”` clause[14], which corresponds to their repository *README.md* files and documentation on https://docs.rs/. However, the *crypto-primes/Cargo.toml* manifest file is missing this clause and the corresponding *README.md* and documentation are silent. As a result, there is no guidance/restrictions on the minimum toolchain version when using this code.

The *crypto-bigint/Cargo.toml* manifest file specifies a dependency on `subtle` version `2.4`. The latest version of this dependency is `2.5` which is approximately 3 months old.

The *crypto-primes/Cargo.toml* manifest file specifies a dependency on `rug` version `1.18`. The latest version of this dependency is `1.19.2` which is approximately 3 months old.

The *elliptic-curves/k256/Cargo.toml* manifest file specifies a dependency on `signature` version 2. This does not include a minor or patch value which is inconsistent with the others.

Other dependencies are essentially current. Note that `cargo audit` did not find any vulnerable (functional) dependencies.

Recommendation

Include the `[package] rust-version = “1.65”` clause into the *crypto-primes/Cargo.toml* manifest file along with the corresponding *README.md* file and documentation on https://docs.rs/.

Update the noted dependencies to the latest version (including minor value) suitable for deployment. Add a periodic gating milestone to the project plan that involves reviewing all dependencies for outdated or vulnerable versions.

Location

- *RustCrypto/crypto-bigint/Cargo.toml*
- *entropyxyz/crypto-primes/Cargo.toml*
- *RustCrypto/elliptic-curves/k256/Cargo.toml*

14. https://blog.rust-lang.org/2021/10/21/Rust-1.56.0.html#cargo-rust-version

Retest Results

2023-08-21 – Fixed

NCC Group reviewed changes to *crypto-primes/Cargo.toml* present in commit `6b069b6` and observed the addition of `rust-version = "1.65"` on line 10. This is aligned with the recommendation. As such, this finding has been marked 'Fixed'. Entropy notes that the maintenance of dependency patch versions is the responsibility of the end user.

Info

Silent Overflow/Wrapping Condition

Overall Risk	Informational	**Finding ID**	NCC-E008526-QTR
Impact	Undetermined	**Component**	crypto-primes
Exploitability	None	**Category**	Error Reporting
		Status	Fixed

Impact

Execution would continue beyond a silent overflow condition resulting in undesired behavior.

Description

The `maybe_next()` function excerpted below as implemented in the *sieve.rs* source file is intended to optionally return the next non-composite number (with respect to the 2048 small primes) to the sieve iterator's `next()` function. This may involve adding the `self.incr` value to the `base` value as highlighted on line 209 below. An overflow resulting from this addition operation would represent an error condition.

```
203 // Returns the restored `base + incr` if it is not composite (wrt the small primes),
204 // and bumps the increment unconditionally.
205 fn maybe_next(&mut self) -> Option<Uint<L>> {
206     let result = if self.current_is_composite() {
207         None
208     } else {
209         let mut num = self.base.wrapping_add(&self.incr.into());
210         if self.safe_primes {
211             num = (num << 1) | Uint::<L>::ONE;
212         }
213         Some(num)
214     };
215 
216     self.incr += 2;
217     result
218 }
```

The above highlighted `wrapping_add()` function will wrap on overflow and thus silently suppress the error condition.

A second instance of this observation is present on line 151 of the same source file. A third instance of this (involving wrapping subtraction) is present on line 105 of *miller-rabin.rs*.

The current code does not appear to be able to trigger this condition, so this finding is marked 'Informational'. However, if the code were to evolve in future, this error condition may become reachable. Additionally, note that the iterator is available to the external user.

Recommendation

Consider detecting this error condition and handling appropriately. This may involve A) a `checked_add()` operation, B) an `adc()` operation followed by an assertion, or C) adding a code comment documenting the potential for a silent error.

Location

- *entropyxyz/crypto-primes/src/hazmat/sieve.rs*
- *entropyxyz/crypto-primes/src/hazmat/miller_rabin.rs*

Retest Results

2023-08-02 – Fixed

NCC Group reviewed changes to *crypto-primes/src/hazmat/sieve.rs* present in commit `7d95cf6` and observed the use of `checked_add()` on line 221 (originally line 209). This is aligned with the recommendation. As such, this finding has been marked ‘Fixed’.

Info `random_mod` Tests Expect Incorrect Behavior

Overall Risk	Informational	**Finding ID**	NCC-E008526-NEW
Impact	None	**Component**	crypto-bigint
Exploitability	None	**Category**	Other
		Status	Fixed

Impact

Existing test suites will fail if run on different valid inputs, suggesting that test coverage may not be as complete as intended.

Description

The `crypto_bigint` crate provides the function `random_mod()` to "generate a cryptographically secure random `Uint` which is less than a given modulus". The provided function appears to be correct, but corresponding test will not pass universally; see *src/uint/rand.rs*:

```rust
    #[test]
    fn random_mod() {
        let mut rng = rand_chacha::ChaCha8Rng::seed_from_u64(1);

        // Ensure `random_mod` runs in a reasonable amount of time
        let modulus = NonZero::new(U256::from(42u8)).unwrap();
        let res = U256::random_mod(&mut rng, &modulus);

        // Sanity check that the return value isn't zero
        assert_ne!(res, U256::ZERO);

        // Ensure `random_mod` runs in a reasonable amount of time
        // when the modulus is larger than 1 limb
        let modulus = NonZero::new(U256::from(0x10000000000000001u128)).unwrap();
        let res = U256::random_mod(&mut rng, &modulus);

        // Sanity check that the return value isn't zero
        assert_ne!(res, U256::ZERO);
    }
```

The test seeds a random number generator with the fixed seed `1` and proceeds to generate a random value modulo `42`. The highlighted assertion enforces that the resulting random number is non-zero. However, zero is a valid output of the `random_mod()` function, and the result is expected to be zero with probability 1 in 42, or approximately 2.3% of the time. Because a fixed seed is used, this condition is not currently hit. The second test case has a similar issue but will fail with a substantially lower probability.

The issue can be triggered by wrapping the test case in a loop until the randomly chosen result is `0`:

```rust
    fn random_mod() {
        let mut rng = rand_chacha::ChaCha8Rng::seed_from_u64(1);
        for _ in 1..100 {
           // Ensure `random_mod` runs in a reasonable amount of time
           let modulus = NonZero::new(U256::from(42u8)).unwrap();
           let res = U256::random_mod(&mut rng, &modulus);
```

```
            // Sanity check that the return value isn't zero
            assert_ne!(res, U256::ZERO);
        }
    }
```

Running the above results in the following test failure, despite the `random_mod()` behavior being correct:

```
running 1 test
thread 'uint::rand::tests::random_mod' panicked at 'assertion failed: `(left != right)`
  left: `Uint(0x0000000000000000000000000000000000000000000000000000000000000000)`,
 right: `Uint(0x0000000000000000000000000000000000000000000000000000000000000000)`',
 ↳ src\uint\rand.rs:67:13
```

It is not clear what the referenced sanity check is intended to enforce, as a zero result is valid. The modulus cannot be zero, and the `NonZero` trait enforces this on creation, but it might make sense to sanity check this in a test where the modulus is generated randomly. Given that the test is currently incorrect, but the issue is not triggered in the current configuration, the test may not be exercising the underlying code sufficiently.

Recommendation

- Remove the invalid check that the result is `0`, or revise it to test a correct property.
- Consider iterating the test over multiple random values instead of a single sample with a fixed seed.

Location

src/uint/rand.rs

Retest Results

2023-08-02 – Fixed

As part of PR #256, commit `be9aa47`, the affected tests were revised to check that the result of `random_mod()` falls within the appropriate range, instead of checking that the result is non-zero. This revision better reflects the intended behavior of the function. As such, this finding has been marked 'Fixed'.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	

Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Appendix: Project Notes and Observations

This informal section contains notes and observations generated throughout the engagement. The content below may be relevant to code improvement opportunities, observations that were noted but not necessarily in scope, and providing supporting information for subsequent analysis. The material is not intended to be comprehensive.

1.0 The *crypto-bigint* Repository

1.1 Code Organization

The *src/boxed* subdirectory contains a relatively small amount of code for fixed-precision heap-allocated unsigned integers (which will wrap rather than grow). Similarly, the *src/limb* subdirectory contains code supporting arithmetic and bitwise logical operations on individual CPU word-sized integers called limbs; checked and unchecked variants are implemented and arithmetic may either wrap or saturate.

The *src/uint* subdirectory contains the bulk of functionality supporting big integers. For modular operations in *unit/modular*, the odd modulus may be fixed at compile-time or calculated during run-time; this is supported by `impl_modulus!` then `const_residue!` macros. Standard arithmetic and bitwise operations are implemented, along with several modular inverse, random, resize and encoding functions.

1.2 Overflow vs Underflow

The term overflow is used to refer to integer operations whose result does not fit into the allocated amount of memory. An overflowing operation in Rust "wraps around" to one that will fit in the allocated type. Wrapping addition on a type `T` will overflow from `T::MAX` to `T::MIN`, and wrapping subtraction will overflow from `T::MIN` to `T::MAX`.

The library was observed to use the term "underflow", both in comments and in code, to refer to integer overflow in the negative direction. While the meaning is understood in context, underflow is most commonly used to refer to a loss of precision when computing over small floating point values. For example, Wikipedia provides the following:

> Storing values that are too low in an integer variable (e.g., attempting to store −1 in an unsigned integer) is properly referred to as integer overflow, or more broadly, integer wraparound. The term underflow normally refers to floating point numbers only, which is a separate issue. It is not possible in most floating-point designs to store a too-low value, as usually they are signed and have a negative infinity value.[15]

1.3 Non-Constant-Time `Cmp` in `Ord` Trait

The `Limb` and `Uint` types implement the `Cmp` operator, which returns one of `Ordering::Less`, `Ordering::Greater`, or `Ordering::Equal`. As seen in the following two implementations, the selection of the output value uses a `match` statement or a series of conditional statements whose runtime will depend on the relative values of the left hand and right hand operands; see *limb/cmp.rs*:

```rust
impl Ord for Limb {
    fn cmp(&self, other: &Self) -> Ordering {
        let mut n = 0i8;
        n -= self.ct_lt(other).unwrap_u8() as i8;
        n += self.ct_gt(other).unwrap_u8() as i8;

        match n {
```

15. https://en.wikipedia.org/wiki/Arithmetic_underflow

```
            -1 => Ordering::Less,
            1 => Ordering::Greater,
            _ => {
                debug_assert_eq!(n, 0);
                debug_assert!(bool::from(self.ct_eq(other)));
                Ordering::Equal
            }
        }
    }
}
```

As well as *uint/cmp.rs*:

```
impl<const LIMBS: usize> Ord for Uint<LIMBS> {
    fn cmp(&self, other: &Self) -> Ordering {
        let is_lt = self.ct_lt(other);
        let is_eq = self.ct_eq(other);

        if is_lt.into() {
            Ordering::Less
        } else if is_eq.into() {
            Ordering::Equal
        } else {
            Ordering::Greater
        }
    }
}
```

The above is not necessarily an issue, as the public interface to the above is provided via a function marked `_vartime` in *limb/cmp.rs*:

```
    /// Perform a comparison of the inner value in variable-time.
    ///
    /// Note that the [`PartialOrd`] and [`Ord`] impls wrap constant-time
    /// comparisons using the `subtle` crate.
    pub fn cmp_vartime(&self, other: &Self) -> Ordering {
        self.0.cmp(&other.0)
    }
```

Similarly, the `inv_mod2k()` function in the *src/uint/inv_mod.rs* source file is constant-time in `self` but not in `k`. This has been addressed in pull 263. Note that the following function `inv_odd_mod_bounded()` in the same source file is variable time in `bits` and `modulus_bits`, but is documented as such.

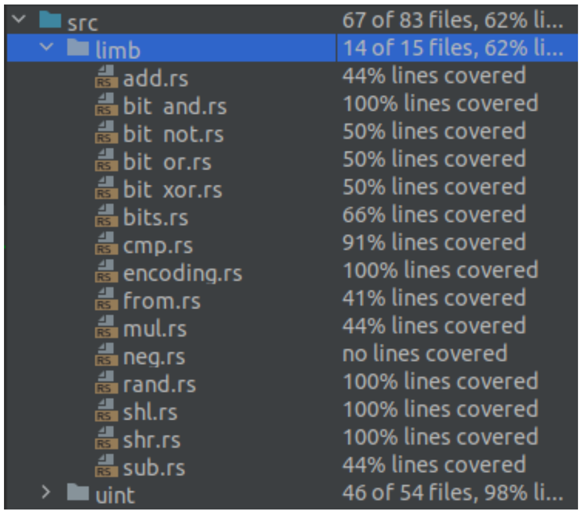
No finding was created for the above non-constant-time code instances, but their existence is being documented as a potential risk for accidental misuse within the crate in the future.

1.4 Debug Mode

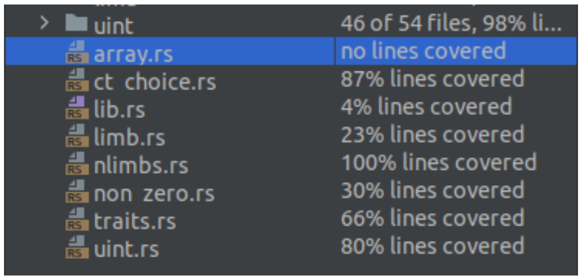
The library contains several `debug_assert!` macros which may not enforce constant-time operations. Indeed, several instances explicitly utilize `_vartime` operations or `PartialOrd` comparisons which, as described in the previous subsection, are not constant time. It is likely that such choices are intentional, and the library does not aim to provide constant-time behavior in debug mode; however, it may be prudent to make this assumption explicit in the documentation to ensure the library is used safely.

1.5 Test Coverage

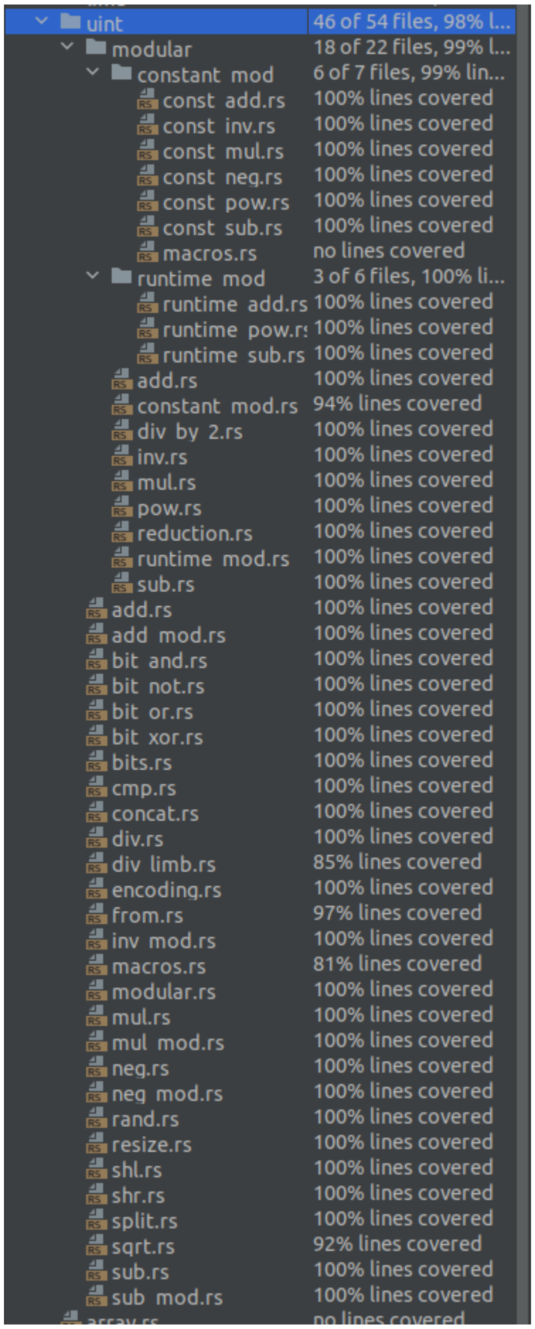
Overall unit-test coverage is robust, though several exceptions can be seen below. The repository does not have a CI/CD test coverage workflow.



```
⌄ src                     67 of 83 files, 62% li...
  ⌄ limb                  14 of 15 files, 62% li...
      add.rs              44% lines covered
      bit and.rs          100% lines covered
      bit not.rs          50% lines covered
      bit or.rs           50% lines covered
      bit xor.rs          50% lines covered
      bits.rs             66% lines covered
      cmp.rs              91% lines covered
      encoding.rs         100% lines covered
      from.rs             41% lines covered
      mul.rs              44% lines covered
      neg.rs              no lines covered
      rand.rs             100% lines covered
      shl.rs              100% lines covered
      shr.rs              100% lines covered
      sub.rs              44% lines covered
  › uint                  46 of 54 files, 98% li...
```



```
  › uint                  46 of 54 files, 98% li...
    array.rs              no lines covered
    ct choice.rs          87% lines covered
    lib.rs                4% lines covered
    limb.rs               23% lines covered
    nlimbs.rs             100% lines covered
    non zero.rs           30% lines covered
    traits.rs             66% lines covered
    uint.rs               80% lines covered
```



```
⌄ uint                        46 of 54 files, 98% l...
  ⌄ modular                   18 of 22 files, 99% l...
    ⌄ constant mod            6 of 7 files, 99% lin...
        const add.rs          100% lines covered
        const inv.rs          100% lines covered
        const mul.rs          100% lines covered
        const neg.rs          100% lines covered
        const pow.rs          100% lines covered
        const sub.rs          100% lines covered
        macros.rs             no lines covered
    ⌄ runtime mod             3 of 6 files, 100% li...
        runtime add.rs        100% lines covered
        runtime pow.r:        100% lines covered
        runtime sub.rs        100% lines covered
      add.rs                  100% lines covered
      constant mod.rs         94% lines covered
      div by 2.rs             100% lines covered
      inv.rs                  100% lines covered
      mul.rs                  100% lines covered
      pow.rs                  100% lines covered
      reduction.rs            100% lines covered
      runtime mod.rs          100% lines covered
      sub.rs                  100% lines covered
    add.rs                    100% lines covered
    add mod.rs                100% lines covered
    bit and.rs                100% lines covered
    bit not.rs                100% lines covered
    bit or.rs                 100% lines covered
    bit xor.rs                100% lines covered
    bits.rs                   100% lines covered
    cmp.rs                    100% lines covered
    concat.rs                 100% lines covered
    div.rs                    100% lines covered
    div limb.rs               85% lines covered
    encoding.rs               100% lines covered
    from.rs                   97% lines covered
    inv mod.rs                100% lines covered
    macros.rs                 81% lines covered
    modular.rs                100% lines covered
    mul.rs                    100% lines covered
    mul mod.rs                100% lines covered
    neg.rs                    100% lines covered
    neg mod.rs                100% lines covered
    rand.rs                   100% lines covered
    resize.rs                 100% lines covered
    shl.rs                    100% lines covered
    shr.rs                    100% lines covered
    split.rs                  100% lines covered
    sqrt.rs                   92% lines covered
    sub.rs                    100% lines covered
    sub mod.rs                100% lines covered
  array.rs                    no lines covered
```

2.0 The *crypto-primes* Repository

The *crypto-primes* crate exposes a straightforward API[16] consisting of:

1. The `generate_prime_with_rng()`, `generate_safe_prime_with_rng()`, `is_prime_with_rng` and `is_safe_prime_with_rng()` functions.

16. https://docs.rs/crypto-primes/latest/crypto_primes/index.html

2. The `generate_prime()`, `generate_safe_prime()`, `is_prime()` and `is_safe_prime()` functions, which are essentially wrappers that call the above corresponding functions with `OsRng` utilized as the random number generator.

Additionally, the library exposes functionality under the `hazmat`[17] qualifier. Potential primes are of type `Uint<L>` from the *crypto-bigint* crate. The `generate*` functions allow the target bit length to be optionally specified, where `None` will correspond to the full length of the returned value `Uint<L>`. The small size and targeted nature of the APIs are conducive to describing the general execution flow below.

The `generate_prime()` function implemented in the *presets.rs* source file simply calls `generate_prime_with_rng()` with `OsRng`. The latter function first confirms the target bit length (if specified) is larger than `2` before entering a (near) infinite `loop{}`. This loop involves generating a random odd unsigned integer, setting that integer as the start value for a new sieve iterator which will generate values not divisible by the first 2048 small primes, followed by another inner `for{}` loop over all returned sieve values/iterations. Each of these latter values is tested for primality via `is_prime_with_rng()` and the enclosing function exits while returning this value. If the sieve becomes 'empty' as it reaches a value beyond the target bit length, the (near) infinite loop is repeated starting with another random odd integer. The only way the function exits is with a value that tests prime.

The `generate_safe_prime()` functionality has a matching structure, but checks the target bit length is larger than `3`, uses an extra flag when creating the sieve and uses `is_safe_prime_with_rng()` for primality testing.

The `is_prime()` function similarly calls the `is_prime_with_rng()` function, which returns `true` for an input of `2` then validates an odd input before calling `_is_prime_with_rng()`. This latter function performs a Miller-Rabin base-2 test, a **Strong** Lucas test and a final Miller-Rabin test on a random base. When a composite number is detected, the function returns `false` immediately; when the function successfully completes, it returns `true`.

Note that all non-hazmat API are hardcoded to involve a **Strong** Lucas test. As such, the execution description content below will primarily focus on this case, with the other cases considered separately.

2.1 *sieve.rs*

The `random_odd_uint()` function implemented in the *sieve.rs* source file first tests that the target bit length is non-zero and not larger than the output type `Uint<L>`. It then generates a full-width random number and optionally shifts it right to trim to the desired bit length. Both the LSB and MSB (based on target bit length) is set and the result returned. The potential to return a fixed value for bit lengths of 1 or 2 is as intended.

The `Sieve::new()` function implemented in the *sieve.rs* source file performs some input validation (see finding "Missing Minimum `max_bit_length` Check") and adjusts the bit length and starting value based on the `safe_primes` flag. The `produces_nothing` flag is calculated based on input corner cases and stored in the struct. Similarly, the condition of starting from a `base` of `2` or less is handled by setting `base` to `3` and `starts_from_exception` to `true`. An empty (but properly sized) vector of residues is allocated upon function exit.

The `next()` function of the sieve iterator returns `None` if the `produces_nothing` flag has been set, then returns either (a safe) `5` or `2` if the `starts_from_exception` flag is set.

17. https://docs.rs/crypto-primes/latest/crypto_primes/hazmat/index.html

Otherwise a while loop is entered which is controlled by the returned value from `update_residues()`. Inside the loop, the function may return `maybe_next()` or simply continue.

The `update_residues()` function will return `false` when on the last round and `true` when a nonzero `incr_limit` is larger than `incr`. Execution will fall through these cases A) upon initialization, or B) upon a needed increase of bit width. When it does, the base is incremented, the reciprocals of the small primes are recalculated via `ct_div_rem_limb_with_reciprocal()` from the *crypto-bigint* crate. Finally, the `incr_limit` is calculated to stay within current bit width.

The `maybe_next()` function uses `current_is_composite()` to check whether `base` + `incr` is divisible by any of the relevant small primes. If it is, the function returns `None` to the while loop inside `next()`, which will then cycle to the next candidate. If the candidate is 'not' composite, a `Some()` is constructed which holds `base` + `incr` or that value divided by two plus one. The `incr` is updated and result returned.

The `current_is_composite()` function checks for divisibility by all small primes. Any remainder of `0` will be reported as composite. If the `safe_primes` flag is set, an additional check on `r == (d - 1) >> 1` is performed (and potentially reported as composite). If both equality tests fail on all small primes, then `false` is returned (indicating probably prime).

2.2 *hazmat/miller_rabin.rs*

The `new()` function for an instance of the Miller-Rabin test confirms the candidate is odd, and determines the constants necessary for conversion to/from Montgomery form along with `one` and `minus_one`. The values of `s` and `d` are determined such that `candidate - 1 = 2`^{s}` * d`. These values are stored in the struct.

The `test_base_two()` function simply calls `test()` with a base of `2`.

The `test()` function converts the supplied base into Montgomery form and then exponentiates it to the power `d`. The result is tested against ± `1` as an early indication of probable primality. The result (`test`) is further squared `1..s` times, and if A) any instance is one then `Composite` is returned, B) if any instance is minus one then `ProbablyPrime` is returned. If the loop completes, then `Composite` is returned. This aligns with the requirement that either A) $a^{d} = 1$, or $a^{d*2^r} = -1$ for some $0 \le r \le s$.

The `test_random_base()` function confirms the bit length is `3` or larger, constructs a random base less than (between 0 and) candidate-4 then adds `3`, then calls `test()`.

2.3 *lucas.rs*

The `lucas_test()` function, when given the Strong variant as is default, (nearly) follows section 3 of "Strengthening the Baillie-PSW primality test"[18] (which provides good background material covering the original primality test). If the candidate is even, return `Composite`. Otherwise, `p` and `q` are chosen in the `generate()` function using the Selfridge base (method **A** not **A***). The execution path continues after `generate()` is briefly described.

The `generate()` function associated with the Selfridge base initializes a few constants before entering an attempt-limited `loop{}` containing a threshold-triggered square check. The loop calculates the Jacobi symbol and breaks on `MinusOne` while returning the proper

18. *Baillie2021*: https://arxiv.org/abs/2006.14425

`p` and `q`. Alternatively, when the loop continues, it calculates the next `d` and tries again. Note that `q = -1` when n ≡ ±3 (mod 10).

The execution path of `lucas_test()` continues by preparing several values for use in calculating the Lucas sequence next following *Baillie2021*. The individual bits of `d` are iterated through while calculating U_{2k} and V_{2k} per equations 13 and 14 respectively, while optionally calculating U_{k+1} and V_{k+1} per equations 16 and 17 respectively (as needed). Condition 11 is tested near line 402, while condition 12 requires more calculation performed on lines 439-457. If neither condition succeeds, a `Composite` indicator is returned.

2.4 *jacobi.rs*

The `jacobi_symbol()` function first validates that the lower argument is odd and the upper argument is not `i32::MIN` (this second check is not necessary and has been fixed). A negative upper argument is then handled via Fact 2.146 (ii) and (i) from The Handbook of Applied Cryptography. Both arguments are then normalized to fit into a Word. The `reduce_numerator()` function follows the latter portion of Fact 2.148 (vii). Once the wider `Uint<L>` is the upper argument, a division can be performed. A `loop{}` performing the reduce and swap operations repeats until the numerator becomes 1.

2.5 *gcd.rs*

This source file implements a helper function to calculate the greatest common denominator of a `Uint<L>` and `u32`. The logic is standard and well documented. Note that it is not constant-time with respect to its operands.

2.6 Observations

While the top-level API is targeted, straightforward and documented, there are several subjective gaps.

1. The technique the implementation uses to choose the parameters `P` and `Q` is known as Method **A**. However, it appears the Strong check defined in section 2.4 of "Strengthening the Baillie-PSW Primality Test"[19] and originally from page 22 (1412) of "Lucas Pseudoprimes"[20] favors Method **A***. The former reference suggests they are effectively equivalent. The implementation includes code for Method **A***.
2. FIPS 186-5[21] places the focus of primality testing on the number of Miller-Rabin tests with Lucas tests in the background – this may make the implementation non-compliant.

The other checks were reviewed separately for compliance to expectations. The `LucasCheck::ExtraStrong` variant checks A) that any of V_{d*2^r} == 0 for 0 ≤ r < s on lines 439-457, *or* B) U_d == 0 and V_d == ±2 on line 418. The `LucasCheck::AlmostExtraStrong` variant checks A) that any of V_{d*2^r} == 0 for 0 ≤ r < s on lines 439-457 *or* B) V_d == ±2 on line 425. The `LucasCheck::LucasV` variant checks If V_{n+1} == 2 * Q on line 465. Note that the terms supporting V_{d*2^r} == 0 for 0 ≤ r < s are still calculated, but ignored. Since prime generation is a probabilistic process, constant-time operation is not exhibited.

3.0 The *elliptic-curves/k256* Repository

The material here is describe from the perspective of both A) base functionality, and then B) scenario-driven execution flow.

19. https://arxiv.org/pdf/2006.14425.pdf
20. http://mpqs.free.fr/LucasPseudoprimes.pdf
21. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf

3.1 *arithmetic/field/**

The *field_5x52.rs* and *field_10x26.rs* source files implement the field elements on a 5-limb 52/48/.../48 struct and 10-limb 22/26/.../26 struct respectively. Each follows the approach in the *bitcoin-core/secp256k1/src/field_** source files. These files include constants, several (precise) serdes functions, constant-time arithmetic, several logical tests, and a variety of functions to handle the lower-level limb element discontinuities (e.g., `normalize_weak()` and `normalize()`). Note that functions such as `add()` do not perform modular reduction on their results (which in this instance arguably contradicts its code comment on line 253 of *field_5x52.rs*) – this is handled at the enclosing scope. Additionally, only a few functions require constant-time `CtOption<Self>` or `Choice`.

The *field_impl.rs* source file contains a debug layer for checking lazy reductions that largely corresponds to and wraps the functionality described above.

3.2 *arithmetic/scalar/**

The *wide64.rs* and *wide32.rs* source files implement functionality for wide scalars on 64-bit and 32-bit limbs respectively. Most of the functionality and line count involves multiplication `mul_wide()`, `mul_shift_vartime()`, `muladd()` and `muladd_fast()` that does not perform modular reduction on the result. A separate function `reduce_impl()` performs the reduction. Note that the `ct_less()` function does not utilize constant-time constructs to introduce an optimization barrier; see finding "Minor Timing Leaks in Wide Scalar Arithmetic".

3.3 *arithmetic/field.rs* and *arithmetic/scalar.rs*

These two files wrap the limb-specific functionality described above into a uniform type. Note that the configuration process/style is not consistent between the two files.

In *field.rs* (for example), the `double()`, `pow2k()`, `invert()` and `sqrt()` (using the (p+1)/4-th power of a) functions are implemented, and randomness is rejection sampled. Straightforward implementations of the `Field` and `PrimeField` traits are present, among other more basic supporting traits.

In *scalar.rs* (for example) values are modulo the curve order and follows a similar structure. There are two functions to generate randomness – one is slightly biased but constant time, and the other uses rejection sampling. The latter is used as the default. The `sqrt()` function uses a variant of the Tonelli-Shanks algorithm with a constant number of iterations and conditional selection. There are also two functions to calculate a scalar inversion – one constant time and another marked `_vartime`.

3.4 *arithmetic/affine.rs* and *arithmetic/projective.rs*

These two source files implement functionality for affine and projective points respectively.

The affine `mul()` point is converted to Projective coordinates, multiplied by the scalar, and returned as a Projective result. Affine addition is not implemented. Most of the functionality here is to support helper traits and serdes functionality.

Projective points have similar functionality but also include `add()`, `add_mixed()`, `double()`, `neg()` and `sub()`. The `mul()`-related operations resolve to `lincomb_generic()` implemented in *mul.rs*.

3.5 *arithmetic/hash2curve.rs*

While the code does not indicate the intended version of IRTF CFRG hash-to-curve specification, the `osswu()` function follows the simplified SWU method described in appendix F.2. The `isogeny()` functionality makes use of the *elliptic-curve* (singular) crate with the coefficients sourced from lines 172 - 256 of *hash2curve.rs*.

3.6 *arithmetic/mul.rs*

The first multiplication function `mul()` takes a scalar and a projective point, is implemented for a variety of traits, and ultimately resolves to `lincomb_generic()` which takes arrays of (length 1 in this instance) scalars and projective points. This latter function first maps the array of scalars across the `decompose_scalar()` function.

The `MINUS_LAMBDA`, `MINUS_B1`, `MINUS_B2`, `G1` and `G2` constants were validated against the output of `sage` https://github.com/bitcoin-core/secp256k1/blob/master/sage/gen_split_lambda_constants.sage and the latter four also against https://github.com/bitcoin-core/secp256k1/blob/master/src/scalar_impl.h#L127-L142 among others. The constants are used within the `decompose_scalar()` function.

The `decompose_scalar()` function is 'inspired' by Algorithm 3.4 in the Guide to Elliptic Curve Cryptography (page 127) steps 4 onwards. The `G1` and `G2` constants correspond to the precomputed estimates noted at the top of page 129. The `c1` and `c2` variables in the code include multiplication by `-B1` and `-B2` respectively. The `r1` and `r2` values are calculated and returned. **This would be a good opportunity to insert `debug_assert` validating that `r1 + r2 * lambda == k mod n`, along with the bit length constraints of `r1` and `r2`** (although the latter are checked during radix-16 decomposition). This function appears to operate in constant time.

After decomposing the array of scalars, the `lincomb_generic()` function continues by separating the decomposed `r1` and `r2` elements then mapping the array of projective points across the `endomorphism()` function. This latter function simply multiplies the point's x-coordinate by the `ENDOMORPHISM_BETA` constant. Next, the `r1` and `r2` magnitudes (signs) are tested against `n/2` and the values remapped such that they are below `n/2` (the shorter representation) and the sign of the corresponding points are flipped as needed. The `r1s_c` and `r2s_c` arrays are then mapped across the `Radix16Decomposition::new()` function. The remainder of `lincomb_generic()` is `double()` and `add()`. These latter functions are found in *projective.rs* and are constant time.

Note that `lincom_generic()` is never utilized (nor tested) with more than 2 scalars and 2 points.

The second multiplication function `mul_by_generator()` is somewhat more straightforward. The scalar is decomposed via the same `Radix16Decomposition::new()` function and a lazily initialized lookup table is referenced (containing points derived from the generator). Two accumulators are initialized, the decomposed digits are iterated across, after which a small doubling patch up loop is executed, and the result returned.

3.7 *schnorr/signing.rs* and *schnorr/verifying.rs*

Schnorr signatures are implemented according to BIP 340[22] and implement the `Signer` trait functions `try_sign()` and `try_sign_digest()`, as well as several other lower-level functions supporting different combinations of pre-hashed data and auxiliary randomness. The signature itself is computed in `sign_prehash_with_aux_rand()`. The signature generation process is mostly straightforward, provided the relevant safety checks are performed alongside each computation:

- Constraints on the signing key `d` are enforced by the underlying `NonZeroScalar` type.
- Similarly, constraints on the ephemeral signing key `k` are also enforced by `NonZeroScalar`.

22. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

- When debug assertions are enabled, the function will also ensure that the produced signature will correctly verify prior to returning. This check is not performed otherwise, which is allowed, but remains a recommendation in BIP 340:

> Verifying the signature before leaving the signer prevents random or attacker provoked computation errors. This prevents publishing invalid signatures which may leak information about the secret key. It is recommended but can be omitted if the computation cost is prohibitive.

Signing tests are provided in *k256/src/schnorr.rs* implementing 4 known answer tests from BIP 340.

Corresponding verification functions are provided using the `Verifier` trait, with the actual verification logic implemented in `verify_prehash()`. Validation of public key, and the signature `(r,s)` is enforced by the underlying types, and the necessary checks on the computed signature are correctly enforced. Similar to the signing case, several test cases are provided, including several negative test cases that ensure the following errors are caught:

- Invalid public key,
- `has_even_y(R)` is false,
- negated message and negated `s` value,
- computed signature is even: `has_even_y(R) == true`,
- x-coordinate for `r` is not on the curve or is equal to field size,
- `s` is equal to the curve order.

3.8 *ecdh.rs* and *ecdsa.rs*

The *ecdh.rs* source file provides the *secp256k1* interface to `elliptic_curve` support. **Note that line 43 contains a copy/paste typo referring to `P-256`**.

The *ecdsa.rs* source file contains the `try_sign_prehashed()` function which follows step 4 onwards of FIPS 186-5[23]. Note that line 216 checks for `s==0` but not `r==0`. This is acceptable as the latter is unreachable since there is no valid point with `x=0` or `x=p mod p` as 7 has no root. However, a code comment noting this situation may be useful. This function additionally calculates and returns a small amount of `recoverId` information to support subsequent key recovery. Hashing is done outside of this function.

23. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf