



Threshold ECDSA Cryptography Review

DFINITY USA Research LLC
Version 1.2 – June 14, 2022

©2022 – NCC Group

Prepared by NCC Group Security Services, Inc. for DFINITY USA Research LLC. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Paul Bottinelli
Marie-Sarah Lacharité
Thomas Pornin

Prepared For
Robin Künzler
Raghav Sundaravaradan

1 Executive Summary

Synopsis

In March 2022, DFINITY engaged NCC Group to conduct a security and cryptography review of a threshold ECDSA implementation, which follows a novel approach described in the reference paper entitled “Design and analysis of a distributed ECDSA signing service” and available on the IACR ePrint archive at <https://eprint.iacr.org/2022/506>. The threshold ECDSA protocol will be deployed into the architecture of the Internet Computer. The ability for canisters to perform threshold signature generation and verification will facilitate the integration of the Internet Computer with other blockchains using ECDSA signatures, including Bitcoin and Ethereum.

The project methodology primarily relied upon manual code review supported by dynamic interaction with the test cases, as well as review of the supporting reference paper. This project was delivered by three consultants over 15 person-days of effort. Full source code access was provided.

Following this review, in early May 2022, NCC Group performed a retest of the findings uncovered during the initial engagement. That follow-up engagement also included the review of a short pull request incorporating changes to the underlying encryption scheme.

Scope

NCC Group’s evaluation included the following components from the `ic` repository, available on GitHub at <https://github.com/dfinity/ic/>. The review was performed on commit `a18a6fa14041650e36444d959dc34ec9b23a23b6`, dated Tuesday March 8.

- The crate `tECDSA`, part of the larger DFINITY cryptographic library.
- The traits `IDkgProtocol`, `ThresholdEcdsaSigner`, `ThresholdEcdsaSigVerifier` defined in the file `canister_threshold_sig.rs` and their respective implementations, including the methods implementing the algorithms defined in the paper, such as `create_dealing()`, `sign_share()` or `verify_sig_share()`.
- A number of code pointers suggested by the DFINITY team, including the function `generate_idkg_dealing_encryption_keys()` to generate dealing encryption keys, as well as a number of traits related to the Crypto Service Provider (CSP) and some of their respective implementations in the Vault.

Additionally, the NCC Group team also used the supporting internal draft reference paper “Design and analysis of a distributed ECDSA signing service” dated March 14, 2022.

During the retest, the NCC Group team also reviewed a pull request ([CRP-1455](#)) that added a zero-knowledge proof to the underlying MEGa encryption and decryption procedures in order to closely follow the reference paper.

Limitations

Good coverage of the in-scope elements was achieved within the given time frame. However, the CSP and Vault implementations are complex components whose functionalities extend past the usage of the threshold ECDSA implementation. As such, they were reviewed in the context of the `tECDSA` library and their integration with the rest of the IC architecture was not investigated in great depth, due to the time-boxed nature of the engagement. Additionally, the development and implementation of state-of-the-art cryptographic algorithms may have subtle issues that a time-boxed review may not necessarily uncover.



Key Findings

The assessment did not uncover any critical or high severity findings. Among the five findings reported, the reviewers identified a common theme; three findings were related to side-channel leakage:

- [Non-Constant-Time Check for Duplicate Scalars.](#)
- [Conditional Assignment Is Not Constant-Time.](#)
- [Square Root Extraction Leaks Input Validity.](#)

A section containing engagement notes is provided in [Engagement Notes](#).

After retesting, NCC Group found that the majority of the findings had been addressed. Out of a total of five (5) original findings, four (4) were marked as *Fixed* while one (1) informational finding was marked as *Partially Fixed*. Additionally, the NCC Group team noted that DFINITY diligently addressed a majority of the observations presented in the informational [Engagement Notes](#) section.

Strategic Recommendations

Pay careful attention to side-channel leakage and to the mitigations implemented, particularly since compiler updates or new target platforms may render some of the mitigations obsolete. Consider writing more end-to-end tests exercising the threshold ECDSA functionalities, including comprehensive negative tests.

Finally, consider updating the implementation to more closely follow the reference paper, including defining and naming structures that mimic the ones defined in the paper. For example, the algorithms implemented could be annotated with direct references to the paper, which would help future reviewers and drive community adoption of this novel algorithm.



2 Dashboard

Target Data

Name	Threshold ECDSA
Type	Cryptographic library and calling functions
Platforms	Rust Implementation
Environment	Local




Engagement Data

Type	Cryptography Review
Method	Code-assisted
Dates	2022-03-07 to 2022-03-18
Consultants	3
Level of Effort	15 person-days + 5 person-days for the retest




Targets

tecdsa	Cryptographic library containing the threshold ECDSA implementation: https://github.com/dfinity/ic/tree/master/rs/crypto/internal/crypto_lib/threshold_sig/tecdsa .
DFINITY's IC Rust repository	General Rust repository which includes some of the traits (i.e. IDkgProtocol, ThresholdEcdsaSigner, and ThresholdEcdsaSigVerifier) under review: https://github.com/dfinity/ic/blob/master/rs .

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	2 
Low issues	2 
Informational issues	1 
Total issues	5

Category Breakdown

Cryptography	3 
Data Validation	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
Conditional Assignment Is Not Constant-Time	Fixed	P4M	Medium
Non-Constant-Time Check for Duplicate Scalars	Fixed	LB4	Medium
Square Root Extraction Leaks Input Validity	Fixed	TBP	Low
MEGa Ciphertext's verify_is() Function Does Not Check Ephemeral Key's Curve	Fixed	WXX	Low
Outdated Dependencies	Partially Fixed	T4U	Info



4 Finding Details

Medium

Conditional Assignment Is Not Constant-Time

Overall Risk Medium
Impact Medium
Exploitability Low

Finding ID NCC-E003936-P4M
Category Cryptography
Status Fixed

Impact

Attackers may infer the value of the control Boolean from timing-based side channels.

Description

The `ct_assign()` function is defined by the `derive_field_element()` macro, as part of an implementation of a prime order finite field. The role of this function is to copy a source value (`other`) into a field element variable (`self`), conditionally on a control value (`assign`):

```
/// If assign is true then set self to other
pub fn ct_assign(&mut self, other: &Self, assign: bool) {
    let mask = 0u64.wrapping_sub(assign as u64);

    for i in 0..#limbs {
        self.limbs[i] = (self.limbs[i] & !mask) ^ (other.limbs[i] & mask);
    }
}
```

This function tries to be *constant-time*, i.e. not to allow outsiders from guessing any information on the source inputs, including the Boolean control value, from timing-based side channels. The code shown above computes a 64-bit “mask” value (the all-zero or all-one value, depending on the value of `assign`), then performs Boolean bitwise operations to select, limb by limb, the new value for the target structure. At least, such is the intent.

The reality is different. Looking at the assembly output produced by the Rust compiler on this function (for a 256-bit modulus, with `rustc 1.57.0` on an `x86_64` Linux system), we find this sequence:

```
testl  %edx, %edx
je     .LBB2_2
movups (%rsi), %xmm0
movups 16(%rsi), %xmm1
movups %xmm1, 16(%rdi)
movups %xmm0, (%rdi)
.LBB2_2:
retq
```

In this code, the `assign` Boolean is obtained in the `edx` register. That value is tested, and, if found equal to 0 (i.e. `false`), a conditional jump (`je`) skips the rest of the function; if the value is not 0, then a simple copy is performed (here, with two 128-bit wide copies using SSE2 registers, since this example was compiled for a 256-bit modulus). In other words, the compiler worked out that the function was really a conditional assignment, and helpfully “optimized” it with a conditional jump and a copy, i.e. a non-constant-time sequence of opcodes. The resulting execution time and memory access pattern will depend on the value the Boolean control value, making it conceptually accessible to attackers through timing-based side channels.



The culprit here is the `bool` type. The compiler knows that a Boolean can only have values `true` and `false`; it thus tries to find what optimizations are possible in the code sequence assuming that the value is one or the other. The compiler then finds that if `assign` is `true`, then `mask` is the all-one value and the loop is a simple copy, which it can unroll and inline, while a value of `false` for `assign` leaves the structure unmodified.

Note: in the current implementation of threshold-ECDSA, `ct_assign()` is invoked only as part of the hash-to-curve implementation, which is not used on private data for the moment. However, `ct_assign()` is a public library function, and the set of potential callers is conceptually unbounded.

Recommendation

The case of Booleans is a special case of *range analysis*, in which the compiler maintains a notion of the possible values of a given variable as a range. When the range is so short that only a few values are possible, the compiler may switch to systematic exploration and use unexpected conditional jumps, as demonstrated here. To avoid such issues, the following rules are recommended:

- Do not use `bool` for secret values. Secret Booleans should be held in an unsigned integer type such as `u64`.
- Do not use `1` for `true`, but a larger value, to obtain a larger “range” (in terms of range analysis). It is convenient to normalize on `0` for `false` and `0xFFFFFFFFFFFFFFFF` for `true`.

Alternatively, rely on the external `subtle`¹ crate for constant-time operations. This crate internally uses the `u8` type to hold secret Booleans, and has [additional mitigations](#) to avoid overzealous compiler optimizations.

Location

[tecdsa/fe-derive/src/lib.rs](#), line 278

Retest Results

2022-05-03 – Fixed

As part of commit [34703fad](#), the `ct_assign()` function was updated according to the recommendation. The function changed the type of the `assign` parameter to a `subtle::Choice`, and now uses the `conditional_select()` function, both provided by the `subtle` crate, as can be seen in the code excerpt below.

```
/// If assign is true then set self to other
pub fn ct_assign(&mut self, other: &Self, assign: subtle::Choice) {
    use subtle::ConditionallySelectable;

    for i in 0..#limbs {
        self.limbs[i] = u64::conditional_select(&self.limbs[i], &other.limbs[i], assign);
    }
}
```

As such, the finding was marked as “Fixed”.

1. <https://crates.io/crates/subtle>



Non-Constant-Time Check for Duplicate Scalars

Overall Risk Medium

Impact Medium

Exploitability Low

Finding ID NCC-E003936-LB4

Category Cryptography

Status Fixed

Impact

Information on the value of the involved secret scalars leaks through timing-based side channels, even in the normal situation where there is no duplicate.

Description

The `contains_duplicates()` function verifies that a given list of elliptic curve scalars are all different from each other:

```
pub(crate) fn contains_duplicates(scalars: &[EccScalar]) -> bool {
    let mut set = std::collections::HashSet::new();

    // This function is only used in cases where we need to exclude duplicates
    // and will immediately return an error, so an early exit (leaking if there
    // are duplicates or not) does not have implications wrt side channels.
    for scalar in scalars {
        if !set.insert(scalar.serialize()) {
            return true;
        }
    }

    false
}
```

The comment indicates that under normal conditions, there will be no duplicate, and if there is, then the duplicate will be excluded and the fact that it happened does not yield any usable information to attackers, thereby making the early exit innocuous. However, even under normal conditions, some extra information on the involved scalars leaks through timing-based side channels:

- The implementation of `HashSet` uses the default hashing and equality functions on the values, which here are encoded scalars and have type `Vec<u8>`. The equality comparison on the two such vectors of bytes is *not* constant-time since it exits early, as soon as a byte difference is found.
- `HashSet` uses a hashtable, accumulating values in “buckets” (lists) indexed by the (truncated) hash value; the memory access pattern when adding a new value into the set will depend on the number of values which already fell into the same bucket, which depends on the hash values computed over previous values. The hash function used is non-cryptographic and any information on its output is liable to be translated into algebraic information on the scalar.

The `contains_duplicate()` function is not public. In the threshold ECDSA implementation, it is called from one place, which is the `at_value()` function that performs Lagrange interpolation on polynomials.

Recommendation

There are two main ways to remove the side-channel leaks reported here:

- The simple way is to replace the `HashSet` with pair-wise constant-time comparisons on the scalars. It will unfortunately require a number of comparisons quadratic in the number of scalars, which may have a high cost if used on a large list of scalars.
- Another way is to generate a random, transient symmetric key K , then use K to compute HMAC/SHA-256 over each serialized scalar, and then use the `HashSet` on the HMAC outputs.

With such methods, the fact that there is a duplicate will still leak (but this is innocuous in the current use of the function), but no otherwise usable information will leak to outsiders through timing-based side channels.

Location

[tecdsa/src/group.rs](#), line 696

Retest Results

2022-05-02 – Fixed

As part of commit [69f47b7d](#), the code was updated to clearly indicate that this non constant-time check was used exclusively on public data.

As such, the finding was marked as “Fixed”.



Square Root Extraction Leaks Input Validity

Overall Risk Low

Finding ID NCC-E003936-TBP

Impact Medium

Category Cryptography

Exploitability Low

Status Fixed

Impact

Outsiders may infer whether a given value was a valid quadratic residue or not, based on timing-based side channels.

Description

The `sqrt()` function, implemented on finite field elements through the `derive_field_element()` macro, computes the square root of a given input. If the input is not a quadratic residue in the field, then the function is documented to return the field element zero:

```
/// Return the square root of self mod p, or zero if no square root exists.
pub fn sqrt(&self) -> Self {
    // For p == 3 (mod 4) square root can be computed using x^(p+1)/4
    let sqrt = self.pow_vartime(&Self::MODULUS_PLUS_1_OVER_4);

    // Check that the result is valid before returning
    if sqrt.square().ct_eq(self) {
        return sqrt;
    }

    Self::zero()
}
```

This calls for two comments:

- If the input is zero, then it is a quadratic residue. Thus, a returned value equal to zero does not necessarily indicate that the input was not a quadratic residue; the caller must also verify whether the input was already zero or not. This test might be missed by the caller, leading to zero wrongfully declared a non-square.
- While the check on the output uses the constant-time `ct_eq()` function, a conditional jump is performed immediately on the output of the test, leading to a different execution time and memory access pattern when the source value is not a quadratic residue. This occurrence is conceptually detectable through timing-based side channels.

The non-constant-time check may lead to exploitable vulnerabilities in some situations, e.g. if the square root call is part of a hash-to-curve process on low-entropy secret input such as a password. In that case, it is normal that square roots may be attempted on values which have probability about 1/2 of being quadratic residues. Information about whether a square root attempt worked or not can help an attacker reduce the cost of dictionary attacks on the low-entropy secret.

Note: In the current tECSA implementation, this `sqrt()` function is not used. However, it is part of the public API, both directly on the defined field element type, and on the `EccFieldElement` wrapper (defined in `src/fe.rs`), making the issue potentially exploitable when the library is used in a larger application.

Recommendation

A constant-time clearing of the result can be obtained in the following way:

```
pub fn sqrt(&self) -> Self {
    // Compute putative square root in r.
    let mut r = self.pow_vartime(&Self::MODULUS_PLUS_1_OVER_4);

    // Check that r^2 yields back the input.
    let t = r.square();
    let mut mm = 0u64;
    for i in 0..#limbs {
        mm |= self.limbs[i] ^ t.limbs[i];
    }
    // mm == 0 if and only if the value r is valid.
    // If mm != 0, then mm or -mm (or both) has its top bit set to 1.
    mm = ((mm | mm.wrapping_neg()) >> 63).wrapping_neg();
    for i in 0..#limbs {
        r.limbs[i] &= mm;
    }
    r
}
```

Additionally, the API may be modified so that the `sqrt()` function returns two values, the value computed above, and an additional `u64` value equal to 0 if the square root failed, or `0xFFFFFFFFFFFFFFFF` otherwise (i.e. the value of `mm` in the code above). Such a modified API would force callers to explicitly consider the failure case, and allow them to use the returned mask for further constant-time operations.

Location

tecdsa/fe-derive/src/lib.rs, line 505

Retest Results

2022-05-03 – Fixed

As part of commit [34703fad](#), the `sqrt()` function was updated and now follows the approach outlined in the recommendation. Specifically, the function no longer performs a conditional jump if the result is invalid, and returns a tuple indicating whether the square root computation was correct. Additionally, equality testing as well as conditional zeroization of the result are now facilitated by the `subtle` crate, as can be seen in the code excerpt below.

```
pub fn sqrt(&self) -> (subtle::Choice, Self) {
    // For p == 3 (mod 4) square root can be computed using x^(p+1)/4
    // though will be nonsense for non quadratic roots.
    let mut sqrt = self.pow_vartime(&Self::MODULUS_PLUS_1_OVER_4);

    let sqrt2 = sqrt.square();

    let is_correct_sqrt = sqrt2.ct_eq(self);

    // zero the result if invalid
    sqrt.ct_assign(&Self::zero(), !is_correct_sqrt);

    (is_correct_sqrt, sqrt)
}
```

As such, the finding was marked as “Fixed”.

MEGa Ciphertext's `verify_is()` Function Does Not Check Ephemeral Key's Curve

Overall Risk	Low	Finding ID	NCC-E003936-WXX
Impact	Low	Category	Data Validation
Exploitability	Undetermined	Status	Fixed

Impact

Incomplete validation of the `ephemeral_key` component of a `MEGaCiphertext` may be contrary to users' expectations and may lead to hard-to-debug errors later in other functions.

Description

MEGa ciphertexts have two components: an `ephemeral_key` (the common component, a single ECC point) and some `ctexts` (the individual components, a vector of ECC scalars for `MEGaCiphertextSingle`, or a vector of pairs of scalars for `MEGaCiphertextPair`). Since both the `EccPoint` and `EccScalar` types have inherent associated elliptic curves (K256 or P256), ciphertexts must be checked to be consistent. A `MEGaCiphertext`'s `verify_is()` function (copied below for reference) appears to provide this validation: it checks that all values in `ctexts` have the same curve type, equal to the function's `curve` parameter. However, it does not perform any validation of the ciphertext's `ephemeral_key` component.

```
141 pub fn verify_is(  
142     &self,  
143     ctype: MEGaCiphertextType,  
144     curve: EccCurveType,  
145 ) -> ThresholdEcdsaResult<> {  
146     let curves_ok = match self {  
147         MEGaCiphertext::Single(c) => c.ctexts.iter().all(|x| x.curve_type() == curve),  
148         MEGaCiphertext::Pairs(c) => c  
149             .ctexts  
150             .iter()  
151             .all(|(x, y)| x.curve_type() == curve && y.curve_type() == curve),  
152     };  
153  
154     if !curves_ok {  
155         return Err(ThresholdEcdsaError::CurveMismatch);  
156     }  
157  
158     if self.ctype() != ctype {  
159         return Err(ThresholdEcdsaError::InconsistentCiphertext);  
160     }  
161  
162     Ok(())  
163 }
```

In particular, it seems that if the function `publicly_verify_dealing()` (`lib.rs`) is passed a `dealing` whose `ciphertext` contains an `ephemeral_key` from a curve other than the one specified in the function's `algorithm_id` parameter, then the function could still return `Ok(())`. The caller could then proceed to call `privately_verify_dealing()`. (Comments indicate that "private verification must be done after the dealing has been publically

verified”). If that function is called, then the `dealing`’s `ciphertext`’s `decrypt()` function will return a `CurveMismatch` error when attempting scalar multiplication of the `ciphertext`’s `ephemeral_key` and the `private_key` (line 426 of `mega.rs` for `MEGaciphertextSingle`, line 516 for `MEGaciphertextPair`). This is the “correct” error, however, a user may be confused since `publicly_verify_dealing()` and `MEGaciphertext.verify_is()` both return `Ok(())`.

Recommendation

- Update the `MEGaciphertext`’s `verify_is()` function in `mega.rs` to check that `self.ephemeral_key`’s curve type is equal to the supplied `curve` parameter.
- Expand testing (in `tests/mega.rs`) to include unit tests for positive and negative cases of `verify_is()`.

Location

[tecdsa/src/mega.rs](#), line 141

Retest Results

2022-05-02 – Fixed

As part of commit [d8605fec](#), the following check was added to the `verify_is()` function:

```
if self.ephemeral_key().curve_type() != curve {  
    return Err(ThresholdEcdsaError::CurveMismatch);  
}
```

As such, the finding was marked as “Fixed”.

Outdated Dependencies

Overall Risk	Informational	Finding ID	NCC-E003936-T4U
Impact	Undetermined	Category	Patching
Exploitability	Low	Status	Partially Fixed

Impact

Unmaintained or deprecated dependencies are unlikely to receive future security updates if a vulnerability is found. Dependencies with known vulnerabilities may provide an advantage to an attacker, as the details of these vulnerabilities are publicly disclosed.

Description

The `cargo-audit` tool automatically scans Rust projects for crates with known security vulnerabilities or warnings. One dependency of the `tecdsa` project gets flagged by this tool because it is unmaintained.

```
Crate:      serde_cbor
Version:    0.11.2
Warning:    unmaintained
Title:      serde_cbor is unmaintained
Date:      2021-08-15
ID:         RUSTSEC-2021-0127
URL:        https://rustsec.org/advisories/RUSTSEC-2021-0127
Dependency tree:
serde_cbor 0.11.2
├── ic-crypto-internal-threshold-sig-ecdsa 0.1.0
└── criterion 0.3.5
    └── ic-crypto-internal-threshold-sig-ecdsa 0.1.0

warning: 1 allowed warning found
```

Additionally, the `cargo outdated` subcommand highlights a number of the dependencies as outdated, see the excerpt below.

Name	Project	Compat	Latest	Kind	Platform
----	-----	-----	-----	----	-----
autocfg->autocfg	1.1.0	---	Removed	Normal	---
block-buffer->block-padding	0.2.1	---	Removed	Normal	---
block-buffer->generic-array	0.14.5	---	Removed	Normal	---
digest->generic-array	0.14.5	---	Removed	Normal	---
fe-derive->num-bigint-dig	0.7.0	---	0.8.1	Normal	---
generic-array->typenum	1.15.0	---	Removed	Normal	---
generic-array->version_check	0.9.4	---	Removed	Build	---
getrandom->cfg-if	1.0.0	---	Removed	Normal	---
getrandom->libc	0.2.120	---	Removed	Normal	cfg(unix)
getrandom->wasi	// <snip>				
getrandom->wasi	// <snip>				
num-bigint-dig->autocfg	0.1.8	---	Removed	Build	---
rand	0.7.3	---	0.8.5	Development	---
rand->getrandom	0.1.16	---	Removed	Normal	---
rand->rand_chacha	0.2.2	---	0.3.1	Normal	// <snip>
rand->rand_core	0.5.1	---	0.6.3	Normal	---
rand->rand_hc	0.2.0	---	Removed	Development	---
rand_chacha	0.2.2	---	0.3.1	Normal	---

rand_chacha->rand_core	0.5.1	---	0.6.3	Normal	---
rand_core	0.5.1	---	0.6.3	Normal	---
rand_core->getrandom	0.1.16	---	0.2.5	Normal	---
rand_core->getrandom	0.1.16	---	Removed	Normal	---
rand_hc->rand_core	0.5.1	---	Removed	Normal	---
sha2	0.9.9	---	0.10.2	Normal	---
sha2->block-buffer	0.9.0	---	Removed	Normal	---
sha2->digest	0.9.0	---	0.10.3	Normal	---
sha2->opaque-debug	0.3.0	---	Removed	Normal	---

The `fe-derive` crate currently also uses a few outdated dependencies.

Name	Project	Compat	Latest	Kind	Platform
----	-----	-----	-----	----	-----
autocfg->autocfg	1.1.0	---	Removed	Normal	---
num-bigint-dig	0.7.0	---	0.8.1	Normal	---
num-bigint-dig->autocfg	0.1.8	---	Removed	Build	---

Recommendation

Update all dependencies and tools to the latest versions recommended for production deployment. Add a gating milestone to the development process that involves reviewing all dependencies for outdated or vulnerable versions.

Location

- [tecdsa/Cargo.toml](#)
- [tecdsa/fe-derive/Cargo.toml](#)

Retest Results

2022-05-03 – Partially Fixed

Some of the dependencies were updated, for example as part of commit [4faa01f2](#).

However, ensuring dependencies are up-to-date and do not expose known vulnerabilities is a constant battle. For example, a run of the `cargo audit` tool (excerpted below) now highlights the `crossbeam-channel` crate as having been yanked.

```
Crate:      crossbeam-channel
Version:    0.5.3
Warning:    yanked
Dependency tree:
crossbeam-channel 0.5.3
├── rayon-core 1.9.1
│   └── rayon 1.5.1
│       └── criterion 0.3.5
│           └── ic-crypto-internal-threshold-sig-ecdsa 0.1.0
```

Additionally, DFINITY indicated that the unmaintained `serde_cbor` crate would be addressed at a later point, and that the risk was currently accepted.

As such, this finding was marked as “Partially 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.



Rating	Description
Medium	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 Engagement Notes

This informational section contains a selected subset of notes and observations generated during the project. While all security issues have already been presented in the preceding individual findings, the following content presents non security-related observations and additional comments on the reference paper.

Hash to curve

The file `tecdsa/src/hash2curve.rs` implements a number of functions to support encoding or hashing of arbitrary strings to points on an elliptic curve. These algorithms follow the IETF draft [Hashing to Elliptic Curves](#). The NCC Group team noted that the version referred to in the source code (v12) is slightly outdated and set to expire on 20 March 2022 – the current draft is v14. Even though the changes are minimal and do not directly affect the algorithms implemented, it is good practice to monitor updates and use the latest possible published versions.

Additionally, some performance improvements could be implemented in the function `sqr_ratio()`, for the generic prime case. Specifically, that function provides a fast algorithm for curves where $p \equiv 3 \pmod{4}$, but implements a naive algorithm in the general prime case, resulting in relatively slow computations.

```
if curve_type == EccCurveType::P256 || curve_type == EccCurveType::K256 {
    // Fast codepath for curves where p == 3 (mod 4)
    // See https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-12.html#appendix-
    // ↪ F.2.1.2
    // <snip>
} else {
    // Generic but slower codepath for other primes
    let z = EccFieldElement::sswu\_z(curve_type);
    let vinv = v.invert();
    let uov = u.mul(&vinv)?;
    let sqrt_uov = uov.sqrt();
    let uov_is_qr = !sqrt_uov.is_zero();
    let z_uov = z.mul(&uov)?;
    let sqrt_z_uov = z_uov.sqrt();
    Ok((uov_is_qr, cmov(&sqrt_z_uov, &sqrt_uov, uov_is_qr?))
}
```

[Appendix F.2.1.1.](#) of the *Hashing to Elliptic Curves* draft defines a faster variant which applies to any field, while [Appendix F.2.1.3.](#) describes an optimized variant for $p \equiv 5 \pmod{8}$.

Minor discrepancy between paper and implementation

In the function `mega_shared_hash_to_scalars()` in `tecdsa/src/mega.rs`, which implements the hash function/random oracle H_M that outputs elements in the message space for the purpose of encryption, an additional input is given to the random oracle, namely the `dealer_index`:

```
ro.add_usize("dealer_index", dealer_index as usize);
ro.add_usize("recipient_index", recipient_index as usize);
ro.add_bytestring("associated_data", associated_data)?;
ro.add_point("public_key", public_key)?;
ro.add_point("ephemeral_key", ephemeral_key)?;
ro.add_point("shared_secret", shared_secret)?;
```

Since this adds more context to the computation than in the reference paper, the security posture is not diminished. However, consider closely matching function calls between the

implementation and the reference paper, and this for *all* functions that are direct implementations of algorithms specified in the reference paper.

Address TODOs

In [consensus/src/ecdsa/complaints.rs](#), a TODO pertaining to the verification of the number of openings satisfying the threshold is still present. This early exit condition should be implemented.

```
// TODO: check num openings satisfies the threshold
match IDkgProtocol::load_transcript_with_openings(&*self.crypto, transcript, &openings) {
// <snip>
```

Additionally, consider addressing all outstanding TODOs (at least pertaining to the threshold ECDSA implementation) prior to deployment.

Serialization of the point-at-infinity

In usual standards covering elliptic curve points (SEC1, ANSI X9.62, FIPS 186-4...), a curve point (x,y) can be encoded in either compressed or uncompressed formats. The uncompressed format starts with a byte of value 0x04, followed by the encodings of x and y (big-endian). The compressed format starts with a byte of value 0x02 or 0x03 (depending on the least significant bit of y), followed by the encoding of x only. The curve point-at-infinity does not have defined x and y coordinates; its standard encoding is a single byte of value 0x00. The elliptic curve backend crates deviate from the standards in that they encode the point-at-infinity into a sequence of bytes of value 0x00, with the same output length as the encoding of a non-infinity byte. Therefore, a “compressed point-at-infinity” for curve secp256k1 is a sequence of 33 bytes of value 0x00.

The `EccPoint` structure, defined in [tecdsa/src/group.rs](#), includes the `serialize()` and `serialize_uncompress()` functions, that output the compressed and uncompressed encodings of a point, respectively. The `deserialize_any_format()` function performs decoding and accepts both compressed and uncompressed formats, while `deserialize()` accepts only compressed encodings. The test in the latter, though, rejects encodings of the point-at-infinity:

```
pub fn deserialize(curve: EccCurveType, bytes: &[u8]) -> ThresholdEcdsaResult<Self> {
    if bytes.len() != curve.point_bytes() || (bytes[0] != 2 && bytes[0] != 3) {
        return Err(ThresholdEcdsaError::InvalidPoint);
    }

    Self::deserialize_any_format(curve, bytes)
}
```

This function rejects any encoding whose first byte is not equal to 0x02 or 0x03; in particular, it rejects the encoding of the point-at-infinity, which starts with 0x00. This implies that `serialize()` and `deserialize()` are not perfect mirrors of each other: `serialize()` may produce an output that `deserialize()` rejects. Whether encodings of the point-at-infinity should be tolerated at all depends on the protocol, but the discrepancy between the behaviours of `serialize()` and `deserialize()` is a source of confusion that will make ulterior software maintenance harder.

Polynomial output length overestimate

The `Polynomial::mul()` function ([src/poly.rs](#), line 178) computes the product of two polynomials. If the number of coefficients of the two operands are m and n , respectively, then the output should have, in general, $m+n-1$ coefficients. The function includes an extra

step to handle the case when both operands are the null polynomial with no coefficient at all ($m = n = 0$):

```
let n_coefs = std::cmp::max(lhs_coefs + rhs_coefs, 1) - 1;
```

The `max()` call, in that case, avoids the production of `-1`, an integer overflow for the unsigned type `usize`. This expression accurately computes the number of coefficients of the result, *except* in case one of the operand is the null polynomial while the other has at least two coefficients: if $m = 0$ but $n \geq 2$, then `n_coefs` is equal to $n-1$, which is at least 1, while the product computation itself produces no coefficient value at all. The produced polynomial then has more coefficients than it should (but the extra coefficients are correctly set to zero).

This is an edge case that probably has no practical consequence.

Undetected failure mode in polynomial interpolation

The `Polynomial::interpolate()` function (defined in `src/poly.rs()`, starting at line 243) computes, given two sequences of scalars x_i and y_i , a polynomial f such that $f(x_i) = y_i$ for all i . The computation processes pairs (x_i, y_i) one by one, building in parallel the polynomial f , and a helper polynomial g whose roots are exactly equal to the already processed x_i . For the input x_i , the process involves in particular computing $1/g(x_i)$, which raises the question of what happens if $g(x_i)$ happens to be equal to zero. In that case, the function simply ignores that pair (x_i, y_i) .

Mathematically, $g(x_i)$ can be zero only if x_i is equal to a previously processed x_j , (for some $i' < i$). If the corresponding y_i is equal to $y_{j'}$, then this is a spurious duplicate and ignoring it is mathematically correct, though that situation is probably erroneous; if $y_i \neq y_{j'}$, then interpolation is not possible and should fail. In either case, the `interpolate()` function should probably report an error whenever $g(x_i) = 0$, rather than silently ignoring it.

Hash function length check in signatures

In `src/sign.rs`, line 327, an explicit check on the hash function length, with regard to the curve size, is performed:

```
// ECDSA has special rules for converting the hash to a scalar,
// when the hash is larger than the curve order. If this check is
// removed make sure these conversions are implemented, and not
// just doing a reduction mod order using from_bytes_wide
if hashed_message.len() != curve_type.scalar_bytes() {
    return Ok(false);
}
```

This calls for two remarks:

1. This check is in the signature verification function (`ThresholdEcdsaCombinedSigInternal::verify()`) but is not present in the signature generation function (`ThresholdEcdsaCombinedSigInternal::new()`). If the situation might arise one day, then it would probably be better to catch it at signature generation time rather than waiting for verification to fail.
2. The check is not technically complete. The ECDSA rules, specified in ANSI X9.62 and FIPS 186-4, are that the hash output is converted into a scalar by first truncating it to the size of the curve (sub)group order *in bits*, then interpreted as an integer (with unsigned big-endian encoding) and reduced modulo that order. The size *in bytes* returned by `scalar_bytes()` may overestimate the actual size. For instance, if this code were to be used with Curve25519 (using its short Weierstraß representation²), whose

subgroup order is a 253-bit integer, the ECDSA rules would call for truncating the SHA-256 output to 253 bits before modular reduction, a step that the current threshold-ECDSA implementation does not perform; however, the check on line 327 would not detect the issue.

In practice, this has no consequence as long as only curves with orders of size exactly 256 bits are used, such as secp256k1 and secp256r1. However, in the interest of future reliability, it would be appropriate to modify the test to use `scalar_bits()` instead of `scalar_bytes()`, and to include a copy of the same test in the signature generation function.

Potential for side-channel leakage from resizing done by `Polynomial.get_coefficients()`

If the parameter `num_coefficients` is greater than the length of the polynomial's `coefficients` vector, whether an error is returned by `Polynomial.get_coefficients()` will depend on whether the higher coefficients are 0 or not. (See `Polynomial.get_coefficients()` in [poly.rs, line 121](#).) Suppose an adversary can control the value of `num_coefficients` passed to this function and can observe whether the function returns an error or not. Then, they could submit `num_coefficients` with one less than the small number of coefficients and see if an error is returned or not. If not, then they would learn whether the leading coefficient is 0.

Currently, the function is not used in a way that makes this side channel exploitable. This note is simply meant to point out that, if the protocol changes the way it uses this function, such abuse may be possible. If the functionality of truncating upper zero coefficients is not necessary, consider removing it to eliminate this side channel.

Also note that in `SimplePolynomial::create()`, a comment says “The polynomial must have at most `num_coefficients` coefficients”, but this function will succeed if it has more coefficients and the extra coefficients are 0.

No-op ciphertext length check

As part of the Pull Request [CRP-1455 Add PoP to MEGa encryption in Threshold ECDSA](#), a `check_validity()` function was introduced in the implementation of the MEGa ciphertexts, in [mega.rs on line 163](#). This function performs a number of consistency checks, the first one being a length check on the number of ciphertexts and the number of expected recipients, passed as a parameter to that function. This can be seen in the code excerpt provided below:

```
pub fn check_validity(
    &self,
    expected_recipients: usize,
    associated_data: &[u8],
    dealer_index: NodeIndex,
) -> ThresholdEcdsaResult<> {
    if self.ctexts.len() != expected_recipients {
        return Err(ThresholdEcdsaError::InvalidRecipients);
    }
    // <snip>
```

2. <https://datatracker.ietf.org/doc/html/draft-ietf-lwig-curve-representations-23>



In some instances, for example in the `decrypt()` function of the same file, this function is called with the length of the ciphertexts vector as the first argument (the `expected_recipients` argument), as follows:

```
self.check_validity(self.ctexts.len(), associated_data, dealer_index)?;
```

Hence, in that specific case, the first check in `check_validity()` essentially tests that `self.ctexts.len()` is equal to itself. Consider slightly modifying the code to pass in the actual expected ciphertexts length in that case.

Additionally, the NCC Group team noticed that commenting out the call to `self.check_validity()` in the `decrypt()` function (the same one highlighted previously) *did not* result in any of the tests failing. While this does not necessarily indicate a potential security issue, it does highlight that no tests are currently in place to test that functionality.

Minor typos in the reference paper

The page numbers correspond to the paper dated March 18, 2022.

- Page 2. “We an analyze the security” -> “We analyze the security”
- Page 4. “The protocol in [GKSS20] has is somewhat” -> “The protocol in [GKSS20] is somewhat”
- Page 4. “Rather than compare are work” -> “Rather than compare **our** work”
- Page 7. “These variations are more variations in the the forgery attack” -> “These variations are more variations in the forgery attack”
- Page 8. “and also a the execution of a resharing” -> “and also the execution of a resharing”
- Page 8. “Typically, the security of of a threshold” -> “Typically, the security of a threshold”
- Page 14. “which us used to identify” -> “which **is** used to identify”
- Page 17. “Feldman’s VSS scheme [Ped91a] is a variation of Feldman’s” -> “Feldman’s should maybe be replaced with “Pedersen’s”
- Page 17. “which is signature on a message” -> “which is **a** signature on a message”
- Page 19. “Suppose the that during key generation” -> “Suppose that during key generation”
- Page 48. “there is no compelling reason to so” -> “there is no compelling reason to **do** so”
- Page 48. “and other aspects of a will change depending on context” -> and “other aspects of a **dealing** will change depending on context”
- Page 50. “We assuming corresponding batch specifications” -> “We **assume** corresponding batch specifications”



7 Contact Info

The team from NCC Group has the following primary members:

- Paul Bottinelli – Consultant
paul.bottinelli@nccgroup.com
- Marie-Sarah Lacharité – Consultant
marie-sarah.lacharite@nccgroup.com
- Thomas Pornin – Consultant
thomas.pornin@nccgroup.com
- Javed Samuel – Practice Director
javed.samuel@nccgroup.com

The team from DFINITY USA Research LLC has the following primary members:

- Robin Künzler
robin.kunzler@dfinity.org
- Raghav Sundaravaradan
raghav.sundaravaradan@dfinity.org

