Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Mina Multisig

Veridise Inc.

June 15, 2026

Last updated date: Jul 09, 2026

► **Prepared For:**

Raspberry Devs
https://github.com/Raspberry-Devs

► **Prepared By:**

Alp Bassa
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Jul 09, 2026	V2
Jul 01, 2026	V1
Jul 01, 2026	Initial Draft

Contents

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the Raspberry Devs team using AuditHub, enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through AuditHub. AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at `https://audithub.dev/case-studies/`.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of Veridise should be considered official Veridise reports.

`https://veridise.com/audits-archive/`

1 Executive Summary

From Jun. 15, 2026 to Jun. 22, 2026, Raspberry Devs engaged Veridise to conduct a security assessment of their Mina Multisig. The security assessment covered the implementation of integrating the FROST multisignature protocol for Mina signatures. Veridise conducted the assessment over 2 person-weeks, with 2 security analysts reviewing the project over 1 week on commit `1457cdd`. The review strategy involved the Veridise security analysts thoroughly reviewing the code.

Project Summary. The security assessment covered two crates in the Mina Multisig project involved in integrating the FROST signature scheme for consumption by Mina smart contracts.

Flexible Round-Optimized Schnorr Threshold (FROST) is a threshold signature scheme for Schnorr Signatures. It allows several parties to posses shares of a private key and create Schnorr signatures under a single public key. Throughout the signing process the private key is never created, the parties exchange messages and sign with their private key shares to obtain a valid signature for the given public key. There is a threshold `t`, such that less than `t` parties cannot sign, but any set of `t` parties can sign.

In order to adapt Mina signatures for the FROST system, Raspberry Devs implemented a wrapper over the Zcash Foundation's frost-core implementation.

The `frost-bluepallas` crate is the Mina-specific FROST ciphersuite crate built on the Pallas curve and Mina's Poseidon-based hashing conventions. It adapts generic `frost-core` threshold signing to Mina by defining the `BluePallas` ciphersuite, field and group serialization rules, Mina-compatible hash functions, parity normalization for commitments, and typed wrappers for the various steps in FROST. It is the cryptographic layer that makes threshold Schnorr signing behave the way Mina expects, while leaving most distributed signing mechanics delegated to `frost-core`.

The `mina-tx` crate provides the compatibility layer between Mina transaction signing data and the `frost-bluepallas` ciphersuite. It defines `PallasMessage`, the Mina-specific challenge payload that preserves data and configuration needed to reproduce Mina's Schnorr challenge computation, along with logic to hash that payload exactly the way Mina expects. It then bridges that message and the resulting FROST outputs back into Mina-facing types by converting a Mina `TransactionEnvelope` into a `PallasMessage` and translating FROST verifying keys and signatures into Mina transaction signature structures suitable for downstream use.

Code Assessment. The Mina Multisig developers provided the source code of Mina Multisig for the code review.

The source code appears to be mostly original code written by the Mina Multisig developers. It contains some documentation in the form of READMEs and documentation comments on functions and data types. To facilitate the Veridise security analysts' understanding of the code, the Mina Multisig developers shared documentation for the code and the underlying `frost-core` crates that are used.

The source code contained a test suite, which the Veridise security analysts noted contains both unit and integration testing.

Summary of Issues Detected. The security assessment uncovered 6 issues, 1 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-MSIG-VUL-001 describes areas in which point of infinity checks are missing. The Veridise analysts also identified 2 medium-severity issues, including V-MSIG-VUL-002, which details that a malformed custom network message can cause a panic. 3 low-severity issues were also found. Raspberry Devs have fixed all of the identified issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Mina Multisig.

Document intended usage of utilities The `signing_utilities.rs` file contains helper functions for generating signatures while given numerous FROST shares. Although these are currently only used in tests, the `mina-multi-sig-audit-rfp.md` indicates these are high priority functions that implement "end-to-end signing flows used in testing and potentially by consumers". If these functions were to be used by consumers, the pooling of all secret material in one place, the lack of a network and participant authentication, and the hardcoded values embedded in the functions will cause issues. It is recommended to provide clear documentation on if these functions will be used outside of an internal testing context, and fix them if they will be.

Document intended consumers Similar to the above recommendation, it is not very clear if the project is a standalone implementation, or if the project should be treated as a library. Depending on the intended usage, the security analysts recommend reviewing what functions and struct variables are publicly exposed.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Mina Multisig	`1457cdd`	Rust	Mina

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jun. 15–Jun. 22, 2026	Manual	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	2	2	2
Low-Severity Issues	3	3	3
Warning-Severity Issues	0	0	0
Informational-Severity Issues	0	0	0
TOTAL	6	6	6

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Cryptographic Vulnerability	2
Logic Error	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Mina Multisig's `frost-bluepallas` and `mina-tx` source code. During the assessment, the security analysts aimed to answer questions such as:

- Does the project correctly and securely integrate with `frost-core`?
- Are appropriate safeguards in place to prevent exposure or leakage of key material, signing shares, nonces, and other sensitive cryptographic state?
- Does the implementation correctly enforce Mina-specific cryptographic requirements, including canonical encoding, y-coordinate parity constraints, and challenge construction?
- Are public keys and signatures following Mina conventions?
- Is the point at infinity properly handled? Are on-curve-checks properly performed?
- Are expected Mina message semantics during serialization, deserialization, and hashing preserved?
- Are messages uniquely defined such that signers always know what they are signing, and they don't collide with Mina transactions?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved the security analysts performing a line-by-line review of the in-scope code and supporting documentation to evaluate design assumptions and identify logic or access-control flaws.

Scope. The scope of this security assessment is limited to the `frost-bluepallas` and `mina-tx` folders of the source code provided by the Mina Multisig developers, which contains the implementation of the Mina Multisig. Specifically, the following files were in scope:

- `frost-bluepallas/src`
  - errors.rs
  - hasher.rs
  - keys.rs
  - lib.rs
  - negate.rs
  - signing_utilities.rs
- `mina-tx/src`
  - bluepallas_compat.rs
  - pallas_message.rs

Methodology. Veridise security analysts reviewed the reports of previous audits for Mina Multisig, inspected the provided tests, and read the Mina Multisig documentation. They then began a review of the code.

During the security assessment, the Veridise security analysts regularly conversed with the Mina Multisig developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-MSIG-VUL-001	Missing point at infinity checks before . . .	High	Fixed
V-MSIG-VUL-002	Custom network parsing during . . .	Medium	Fixed
V-MSIG-VUL-003	Group commitment computation lacks . . .	Medium	Fixed
V-MSIG-VUL-004	Incorrect SignatureSerialization size	Low	Fixed
V-MSIG-VUL-005	Undesirable message deserialization fallback	Low	Fixed
V-MSIG-VUL-006	Mina signature conversion does not . . .	Low	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MSIG-VUL-001: Missing point at infinity checks before coordinate access

Severity	High	**Commit**	1457cdd
Type	Cryptographic Vulnerabilit	**Status**	Fixed
Location(s)	▶ `frost-bluepallas/src/lib.rs` ▶ `mina-tx/src/pallas_message.rs`		
Confirmed Fix At	mina-multi-sig/pull/198		

Description At several places in the code `x` and `y` coordinates of elliptic curve points are accessed without first checking whether the point is the point at infinity. In arkworks short-Weierstrass affine representation, the point at infinity is represented with `x=0`, `y=0` and `infinity=true`. Therefore accessing the coordinate of points using

```
point.into_affine().x
point.into_affine().y
```

will not fail for the point at infinity and will silently return `0`.

In the usual frost-core flow, rejcetion of identity-points is ensured during group-element serialization and deserialization, and the default challenge computation serializes both the aggregate commitment `R` and the verifying key before hashing them. The codebase overrides the challenge computation to match Mina's hashing and reads affine coordinates directly instead of calling `serialize`. As a result, the corresponding identity check is bypassed, necessitating explicit point-at-infinity checks before using aggregate commitment coordinates, signature commitment coordinates, or public-key coordinates.

In `mina_tx/src/pallas_message.rs` instances of not checking the coordinates exist at the following locations:

- Line 206: `fr_sig.R().into_affine().x`
- Line 263-264: `pub_key_x: pub_key.point().x` and `pub_key_y: pub_key.point().y`
- Line 288: `r.into_affine().x`

In `frost_bluepallas/src/lib.rs` instances of not checking the coordinates exist at the following locations:

- Line 213 and 243: `commit.to_element().into_affine().y.into_bigint().is_even()`: This causes the `is_even()` function to return true for the point at infinity `y`-coordinate.

Impact Accessing a coordinate directly circumvents any checks regarding the point at infinity, and therefore a value may be used which comes from a point that is not on the curve. `x` and `y` coordinates will erroneously be read as `0`.

Recommendation Add checks that the point is not the point at infinity, explicitly rejecting infinity.

Developer Response The developers have added checks to all the listed locations.

4.1.2 V-MSIG-VUL-002: Custom network parsing during deserialization can cause panic

Severity	Medium	**Commit**	1457cdd
Type	Data Validation	**Status**	Fixed
Location(s)	`mina-tx/src/pallas_message.rs:100-139`		
Confirmed Fix At	mina-multi-sig/pull/199		

Description When parsing a custom network inside of the `PallasMessage::deserialize()` function, the third byte of the `input` is interpreted as the length of the custom network string. The code block proceeds to check that the length of `input` is at least `3 + name_len` and proceeds to return `3 + name_len` as the `offset` to continue parsing the `input`. As can be seen below, the `input` is indexed by `offset` when reading the `is_legacy` flag:

```
let (network_id, offset) = match input[1] {
     /// ... VERIDISE ELIDED ...
     2 => {
         let name_len = input[2] as usize;
                    /// ... VERIDISE ELIDED ...
         if input.len() < 3 + name_len {
             return Err(crate::errors::MinaTxError::DeSerializationError(
                 "PallasMessage too short for custom network ID name".into(),
             ));
         }
         /// ... VERIDISE ELIDED ...
         (NetworkId::Custom(name), 3 + name_len)
     }
        /// ... VERIDISE ELIDED ...
 };
let is_legacy = match input[offset] {
    /// ... VERIDISE ELIDED ...
};
```

Snippet 4.1: Snippet from `pallas_message.rs:L100-139`

A malformed `input` with a length of `3 + name_len` will cause this function to panic.

Impact This function is called by `PallasMessage::challenge()`, which is used by `round2::sign()` and therefore may cause a DoS on clients receiving this message if no other length validation is performed.

Recommendation Increase the length validation during custom network parsing by 1 to ensure that attempting to index into an invalid index will not occur.

Developer Response The developers have fixed the off by one error in the length check of custom network parsing.

4.1.3 V-MSIG-VUL-003: Group commitment computation lacks nonzero check

Severity	Medium	**Commit**	1457cdd
Type	Cryptographic Vulnerabilit	**Status**	Fixed
Location(s)	`frost-bluepallas/src/lib.rs`		
Confirmed Fix At	mina-multi-sig/pull/198		

Description In both the `pre_commitment_aggregate()` and `pre_commitment_sign()` functions, the group commitment (`commit`) is calculated and used as follows:

```
1 let commit = compute_group_commitment(signing_package, binding_factor_list)?;
```

Snippet 4.2: Snippet from `lib.rs`:

However, the `commit` is never checked to be nonzero. This means the rest of the FROST protocol may derive challenges from the point of infinity.

Impact When a zero value `commit` is used, challenges may be derived by the point of infinity and construct and/or validate invalid signatures.

Recommendation Check that the computed `commit` value is nonzero. Additionally, implementing the fix for V-MSIG-VUL-001 will prevent a zero value from being generated.

Developer Response The developers have have created a new `group_commitment_affine()` function which calls the underlying `compute_group_commitment()` and checks that the value is nonzero.

4.1.4 V-MSIG-VUL-004: Incorrect SignatureSerialization size

Severity	Low	**Commit**	1457cdd
Type	Logic Error	**Status**	Fixed
Location(s)	`frost-bluepallas/src/lib.rs:169`		
Confirmed Fix At	mina-multi-sig/pull/201		

Description `BluePallas` implements the `Ciphersuite` trait, which defines the `SignatureSerialization` type. As detailed in the frost-core documentation, this length should be of `Group::ElementSerialization + Group::ScalarSerialization` in order to represent the `(R,z)` signature. In this context, that means the byte array should be sized as `65` (`33 + 32`). However, the current size incorrectly uses the `HASH_SIZE` constant, which is 32.

Impact Currently this does not appear to cause signature serialization or verification failures. `frost-core`'s default `Signature::serialize()` and `Signature::deserialize()` implementations determine the expected signature length by serializing the group generator and scalar zero value, which results in the correct length.

However, the ciphersuite contract is incorrect. Any downstream or future generic code that relies on `SignatureSerialization` as the canonical fixed-size signature representation may allocate the wrong buffer size, reject valid 65-byte signatures, truncate signatures, or otherwise produce malformed-signature errors.

Recommendation Fix the type to be sized correctly as `GROUP_SIZE + FIELD_SIZE`.

Developer Response The developers now size the type as `GROUP_SIZE+FIELD_SIZE`.

4.1.5 V-MSIG-VUL-005: Undesirable message deserialization fallback

Severity	Low	**Commit**	1457cdd
Type	Data Validation	**Status**	Fixed
Location(s)	`mina-tx/src/pallas_message.rs:291-292`		
Confirmed Fix At	mina-multi-sig/pull/203		

Description `PallasMessage::challenge()` silently falls back to `from_raw_bytes_default()` when `PallasMessage::deserialize()` fails. This will interpret the `message` bytes directly, and set the `network_id` to `Testnet` and `is_legacy` to true, as seen below:

```
pub fn from_raw_bytes_default(input: &[u8]) -> Self {
    Self {
        input: ROInput::new().append_bytes(input),
        network_id: NetworkId::Testnet,
        is_legacy: true,
    }
}
```

Snippet 4.3: Snippet from 'pallas_message:L48-54

This leads to malformed or non`PallasMessage` inputs as being reinterpreted instead of rejected.

Impact Malformed messages are not rejected, and instead signed/verified under different semantics than the caller assumes. Not only will this behavior be potentially unnoticed by consumers of the ciphersuite, but callers who intended to sign a structure Mina message may accidentally sign a raw-byte Testnet message instead.

Recommendation Guard the deserialization fallback behavior behind `#[cfg(test)]` and return an error for non-test builds. This will prevent the issue for release binaries while allowing the test harness to still run.

Developer Response The developers have guarded the fallback behavior behind a `raw-message-fallback` crate feature. This requires users to explicitly enable this behavior, and the test suite enables this feature.

4.1.6 V-MSIG-VUL-006: Mina signature conversion does not validate y-coordinate parity

Severity	Low	**Commit**	1457cdd
Type	Data Validation	**Status**	Fixed
Location(s)	`mina-tx/src/` ▶ `bluepallas_compat.rs:20` ▶ `pallas_message.rs:203`		
Confirmed Fix At	mina-multi-sig/pull/200		

Description Mina signatures have a standard y-coordinate convention for the Schnorr commitment `R` and only its x-coordinate is stored. The Frost to Mina signature conversion helpers `translate_sig()` in `pallas_message.rs` and `Tryfrom()` in `bluepallas_compat.rs` do not validate that this assumption holds and immediately discard `R.y`, keeping `R.x` without checking the parity of the y-coordinate.

Impact Providing the conversion function with a signature with an `R` that has an odd y-coordinate will result in a potentially invalid Mina signature by retaining only the x-coordinate. This can cause signing workflows to report or return a converted signature that later fails, creating interoperability and availability risk.

Recommendation In helpers for signature conversion check parity of y-coordinate of the Schnorr commitment `R`. If `R.y` is odd, return an explicit conversion error.

Developer Response The developers now check if the `y` coordinate is odd, and return an error or negate when appropriate.