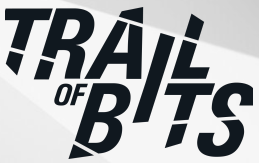

DFINITY Threshold ECDSA Integration and Bitcoin canisters

Security Assessment

July 7, 2022

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Fredrik Dahlgren and Will Song**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits 1

Notices and Remarks 2

Table of Contents 3

Executive Summary 4

Project Summary 6

Project Goals 7

Project Targets 8

Project Coverage 9

Automated Testing 11

Codebase Maturity Evaluation 13

Summary of Findings 16

Detailed Findings 17

1. Lack of validation of signed dealing against original dealing 17

2. The ECDSA payload is not updated if a quadruple fails to complete 20

3. Malicious canisters can exhaust the number of available quadruples 22

4. Aggregated signatures are dropped if their request IDs are not recognized 24

A. Vulnerability Categories 25

B. Code Maturity Categories 27

C. Automated Testing 29

D. Code Quality Recommendations 34

Executive Summary

Engagement Overview

DFINITY engaged Trail of Bits to review the security of the integration of its ECDSA threshold signature scheme with its consensus protocol and Bitcoin canister and adapter. From May 2 to May 13, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered one significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	1
Informational	1
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Denial of Service	2

Notable Findings

A significant flaw that impacts system confidentiality, integrity, or availability is listed below.

- **TOB-DFTECDSA-1**
  By changing a dealing payload before creating the corresponding dealing support, a malicious node could prevent a quadruple from completing. Since signing requests are deterministically matched against quadruples, this issue could allow a single node to block the service of individual signing requests.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 28, 2022	Kickoff call: ECDSA–consensus protocol integration
May 2, 2022	Kickoff call: ECDSA–Bitcoin integration
May 9, 2022	Status update meeting #1
May 16, 2022	Delivery of report draft
May 16, 2022	Report readout meeting
July 7, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the DFINITY ECDSA threshold signature scheme integration. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the ECDSA integration with the consensus protocol follow the specification laid out in the paper “Design and analysis of a distributed ECDSA signing service” [GS22]?
- Does the node correctly validate new protocol artifacts like dealings, dealing supports, and signature shares?
- Are signatures verified and aggregated correctly?
- Could a node generate multiple artifacts of the same type for the same transcript ID?
- Are signing requests deterministically paired with quadruples?
- Could the same quadruple be reused across multiple signing requests?
- Is the nonce re-randomization mechanism implemented correctly?
- Is it possible to exploit the complaints mechanism to obtain openings for honest nodes?
- Is the distributed signing service vulnerable to denial-of-service attacks?
- Does the Bitcoin adapter correctly handle valid messages from the Bitcoin network?
- Does the Bitcoin adapter disconnect upon receiving invalid messages from the Bitcoin network?
- Does the Bitcoin canister correctly handle updates from the adapter?
- Does the Bitcoin canister return the correct information via its API?
- Do the Bitcoin canister and adapter correctly store information about the Bitcoin network?
- Are the Bitcoin canister and adapter implemented correctly?

Project Targets

The engagement involved a review and testing of the targets listed below.

ECDSA Consensus Integration

Repository	`dfinity/ic/rs/consensus`
Version	`6febeeadc3dfefbb85bded262ff999ea8764e8a5`
Type	Rust
Platform	Linux

ECDSA Bitcoin Integration

Repository	`dfinity/ic/rs/bitcoin`
Version	`6febeeadc3dfefbb85bded262ff999ea8764e8a5`
Type	Rust
Platform	Linux

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **Consensus integration.** We manually reviewed the ECDSA pre-signer and signer implementations, focusing on identifying correctness and validation issues. In particular, we made sure that dealers cannot submit multiple dealings for the same transcript and that signers cannot submit multiple signature shares for the same signing request. We also verified that the pre-signer does not accept pre-signature shares from nodes outside a transcript's prescribed set of dealers and that new signature shares are validated against the correct key. Finally, we checked whether it would be possible for a malicious signer to split the network by distributing signature shares for different messages to different nodes.

  We reviewed the ECDSA gossip mechanism to ensure that old and invalid protocol artifacts are purged correctly. We then reviewed the ECDSA payload creation mechanism to ensure that signing requests are matched deterministically with quadruples (both those available and those being created) (as outlined in the appendix of the paper [GS22]). We also ensured that multiple requests cannot be matched against the same quadruple. Finally, we investigated potential denial-of-service attack vectors against signing requests in case a node manages to stop a quadruple from completing.

  We reviewed the quadruple generation and key reshare state machines for correctness. Finally, we reviewed the ECDSA complaints mechanism. In particular, we ensured that both complaints and openings are verified correctly. We also checked that the implementation rejects complaints against honest nodes and that openings are sent only when a valid complaint has been received.

- **Bitcoin integration.** The Bitcoin adapter and canister were reviewed for implementation correctness. This involved a heavy focus on the Bitcoin network interface and the internal representation of the Bitcoin network. Both the Bitcoin canister and adapter received extensive review, which uncovered no significant findings.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The ECDSA payload validation was not completed in time for the review; therefore, it was not covered during the assessment.
- We did not perform an in-depth review of the remote procedure calls (RPCs) between the replica and the adapter. We assume that matching versions would not send invalid messages to each other.
- During this engagement, issues with running Tarpaulin, `llvm-cov`, and `kcov` prevented us from obtaining reliable metrics on the unit test coverage of the codebase. The reason for this is most likely that we did not have access to a Linux environment with access to DFINITY's internal network.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C
Tarpaulin	An open-source code coverage reporting tool that can be integrated with the Cargo build system on x86 Linux architectures	Appendix C

Areas of Focus

Our automated testing and verification work focused on identifying the following issues:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`
- Poor unit and integration test coverage

Test Results

The results of this focused testing are detailed below.

ECDSA consensus integration: Integration of the ECDSA threshold signature scheme with the consensus algorithm

Property	Tool	Result
The project adheres to Rust best practices by fixing code quality issues reported by linters like Clippy.	Clippy	**Passed**
The project's use of panicking functions like `unwrap` and `expect` is limited.	Semgrep	**Passed**
All components of the codebase have sufficient test coverage.	Tarpaulin	**Further Investigation Required**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The consensus integration does not contain much arithmetic. Both the consensus and Bitcoin integrations use custom type wrappers for u64 integers, which are unlikely to overflow.	**Strong**
Auditing	Unexpected and potentially malicious behavior is adequately logged by the ECDSA consensus integration. However, in some cases, the Bitcoin integration could benefit from replacing uses of `unwrap` with `expect`, which would result in more explicit error messages.	**Strong**
Authentication / Access Controls	All privileged operations performed by the consensus integration are signed by the originating node, and we found no issues with how signatures are validated by the implementations of the ECDSA pre-signer, signer, and complaints handlers.	**Strong**
Complexity Management	The codebase is not overly complex, and each component of the consensus integration implements a well-defined and easily reviewable API. This also applies to the ECDSA Bitcoin integration. However, some of the functions related to ECDSA payload generation are very long and would benefit from being broken up into smaller components.	**Satisfactory**
Configuration	The number of quadruples that are created in advance currently defaults to one, which is too low for a	**Moderate**

	production deployment of the system.	
Cryptography and Key Management	Signing keys are shared among all of the nodes in the subnet. The implementation also allows keys to be reshared. This is mainly used if a node either leaves or joins the network, but could also be used to mitigate a partial compromise of the signing key shares.	**Strong**
Data Handling	We found that consensus protocol artifacts related to ECDSA are validated correctly in all but one case. Signed messages are not validated for dealing support shares, which could be used by an attacker to stop signing requests from completing (TOB-DFTECDSA-1).	**Satisfactory**
Decentralization	The implementation of the ECDSA service is decentralized in the sense that no single node has access to either the pre-signature quadruples or the signing key used to sign new messages. However, the service is currently deployed only on a single subnet, which means that the cross-message bandwidth of this single network may limit the number of signing requests that can be serviced by the system.	**Satisfactory**
Documentation	The paper [GS22] provides detailed documentation of the underlying signature protocol and extensive implementation details that are relevant when the protocol is deployed as a service on the Internet Computer. Additionally, we found the source-level documentation for the consensus component very helpful when reviewing the implementation. We also found that the documentation for the Bitcoin integration is sufficient.	**Strong**
Maintenance	Any component of the protocol may be updated as part of a regular update of the replica software.	**Strong**

Memory Safety and Error Handling	The codebase has very explicit error handling mechanisms with descriptively named error enums. This makes it easier to track error conditions throughout the codebase and to ensure that errors are handled correctly. The implementation contains very few calls to panicking functions like `expect` and `unwrap`, and the consensus codebase contains no unsafe code.	**Strong**
Testing and Verification	Because of a compilation issue, we were unable to use Tarpaulin to get reliable test coverage metrics for the consensus crate. We also tried to use `llvm-cov` and `kcov` to generate coverage data, but we were unsuccessful. The reason for this is most likely that we did not have access to a Linux environment with access to DFINITY's internal network. The `bitcoin` crate has a number of unit tests. It is unclear how to improve testing regarding the Bitcoin integration, as it involves multiple pieces of software as well as network data that is not easy to generate on the fly.	**Further Investigation Required**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of validation of signed dealing against original dealing	Data Validation	**Medium**
2	The ECDSA payload is not updated if a quadruple fails to complete	Data Validation	**Low**
3	Malicious canisters can exhaust the number of available quadruples	Denial of Service	**Undetermined**
4	Aggregated signatures are dropped if their request IDs are not recognized	Denial of Service	**Informational**

Detailed Findings

1. Lack of validation of signed dealing against original dealing

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-DFTECDSA-1
Target: `ic/rs/consensus/src/ecdsa/pre_signer.rs`	

Description

The `EcdsaPreSignerImpl::validate_dealing_support` method does not check that the content of signed dealings matches the original dealings. A malicious receiver could exploit this lack of validation by changing the requested height (that is, the `support.content.requested_height` field) or internal data (the `support.content.idk_dealing.internal_dealing_raw` field) before signing the ECDSA dealing. The resulting signature would not be flagged as invalid by the `validate_dealing_support` method but would result in an invalid *aggregated* signature.

After all nodes sign a dealing, the `EcdsaTranscriptBuilderImpl::build_transcript` method checks the signed dealings' content hashes before attempting to aggregate all the dealing support signatures to produce the final aggregated signature. The method logs a warning when the hashes do not agree, but does not otherwise act on signed dealings with different content.

```
let mut content_hash = BTreeSet::new();
for share in &support_shares {
    content_hash.insert(ic_crypto::crypto_hash(&share.content));
}
if content_hash.len() > 1 {
    warn!(
        self.log,
        "Unexpected multi share content: support_shares = {}, content_hash = {}",
        support_shares.len(),
        content_hash.len()
    );
    self.metrics.payload_errors_inc("invalid_content_hash");
}

if let Some(multi_sig) = self.crypto_aggregate_dealing_support(
    transcript_state.transcript_params,
    &support_shares,
```

```
) {
    transcript_state.add_completed_dealing(signed_dealing.content, multi_sig);
}
```

Figure 1.1: ic/rs/consensus/src/ecdsa/pre_signer.rs:1015-1034

The dealing content is added to the set of completed dealings along with the aggregated signature. When the node attempts to create a new transcript from the dealing, the aggregated signature is checked by `IDkgProtocol::create_transcript`. If a malicious receiver changes the content of a dealing before signing it, the resulting invalid aggregated signature would be rejected by this method. In such a case, the `EcdsaTranscriptBuilderImpl` methods `build_transcript` and `get_completed_transcript` would return `None` for the corresponding transcript ID. That is, neither the transcript nor the corresponding quadruple would be completed.

Additionally, since signing requests are deterministically matched against quadruples, including quadruples that are not yet available, this issue could allow a single node to block the service of individual signing requests.

```
pub(crate) fn get_signing_requests<'a>(
    ecdsa_payload: &ecdsa::EcdsaPayload,
    sign_with_ecdsa_contexts: &'a BTreeMap<CallbackId, SignWithEcdsaContext>,
) -> BTreeMap<ecdsa::RequestId, &'a SignWithEcdsaContext> {
    let known_random_ids: BTreeSet<[u8; 32]> = ecdsa_payload
        .iter_request_ids()
        .map(|id| id.pseudo_random_id)
        .collect::<BTreeSet<_>>();
    let mut unassigned_quadruple_ids =
        ecdsa_payload.unassigned_quadruple_ids().collect::<Vec<_>>();
    // sort in reverse order (bigger to smaller).
    unassigned_quadruple_ids.sort_by(|a, b| b.cmp(a));
    let mut new_requests = BTreeMap::new();
    // The following iteration goes through contexts in the order
    // of their keys, which is the callback_id. Therefore we are
    // traversing the requests in the order they were created.
    for context in sign_with_ecdsa_contexts.values() {
        if known_random_ids.contains(context.pseudo_random_id.as_slice()) {
            continue;
        };
        if let Some(quadruple_id) = unassigned_quadruple_ids.pop() {
            let request_id = ecdsa::RequestId {
                quadruple_id,
                pseudo_random_id: context.pseudo_random_id,
            };
            new_requests.insert(request_id, context);
        } else {
            break;
        }
    }
```

```
    new_requests
}
```

Figure 1.2: ic/rs/consensus/src/ecdsa/payload_builder.rs:752-782

Exploit Scenario

A malicious node wants to prevent the signing request SR_i from completing. Assume that the corresponding quadruple, Q_i, is not yet available. The node waits until it receives a dealing corresponding to quadruple Q_i. It generates a support message for the dealing, but before signing the dealing, the malicious node changes the `dealing.idk_dealing.internal_dealing_raw` field. The signature is valid for the updated dealing but not for the original dealing.

The malicious dealing support is gossiped to the other nodes in the network. Since the signature on the dealing support is correct, all nodes move the dealing support to the validated pool. However, when the dealing support signatures are aggregated by the other nodes, the aggregated signature is rejected as invalid, and no new transcript is created for the dealing.

This means that the quadruple Q_i never completes. Since the matching of signing requests to quadruples is deterministic, SR_i is matched with Q_i every time a new ECDSA payload is created. Thus, SR_i is never serviced.

Recommendations

Short term, add validation code in `EcdsaPreSignerImpl::validate_dealing_support` to verify that a signed dealing's content hash is identical to the hash of the original dealing.

Long term, consider whether the BLS multisignature aggregation APIs need to be better documented to ensure that API consumers verify that all individual signatures are over the same message.

2. The ECDSA payload is not updated if a quadruple fails to complete	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-DFTECDSA-2
Target: Consensus integration	

Description

If a transcript fails to complete (as described in TOB-DFTECDSA-1), the corresponding quadruple, Q_i, will also fail to complete. This means that the quadruple ID for Q_i will remain in the `quadruples_in_creation` set until the key is reshared and the set is purged. (Currently, the key is reshared if a node joins or leaves the subnet, which is an uncommon occurrence.) Moreover, if a transcript and the corresponding Q_i fail to complete, so will the corresponding signing request, SR_i, as it is matched deterministically with Q_i.

```
let ecdsa_payload = ecdsa::EcdsaPayload {
    signature_agreements: ecdsa_payload.signature_agreements.clone(),
    ongoing_signatures: ecdsa_payload.ongoing_signatures.clone(),
    available_quadruples: if is_new_key_transcript {
        BTreeMap::new()
    } else {
        ecdsa_payload.available_quadruples.clone()
    },
    quadruples_in_creation: if is_new_key_transcript {
        BTreeMap::new()
    } else {
        ecdsa_payload.quadruples_in_creation.clone()
    },
    uid_generator: ecdsa_payload.uid_generator.clone(),
    idkg_transcripts: BTreeMap::new(),
    ongoing_xnet_reshares: if is_new_key_transcript {
        // This will clear the current ongoing reshares, and
        // the execution requests will be restarted with the
        // new key and different transcript IDs.
        BTreeMap::new()
    } else {
        ecdsa_payload.ongoing_xnet_reshares.clone()
    },
    xnet_reshare_agreements: ecdsa_payload.xnet_reshare_agreements.clone(),
};
```

Figure 2.1: The `quadruples_in_creation` set will be purged only when the key is reshared.

The canister will never be notified that the signing request failed and will be left waiting indefinitely for the corresponding reply from the distributed signing service.

Recommendations

Short term, revise the code so that if a transcript (permanently) fails to complete, the quadruple ID and corresponding transcripts are dropped from the ECDSA payload.

To ensure that a malicious node cannot influence how signing requests are matched with quadruples, revise the code so that it notifies the canister that the signing request failed.

3. Malicious canisters can exhaust the number of available quadruples

Severity: **Undetermined**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-DFTECDSA-3
Target: Consensus integration	

Description

By requesting a large number of signatures, a canister (or set of canisters) could exhaust the number of available quadruples, preventing other signature requests from completing in a timely manner.

The ECDSA payload builder defaults to creating one extra quadruple in `create_data_payload` if there is no ECDSA configuration for the subnet in the registry.

```
let ecdsa_config = registry_client
    .get_ecdsa_config(subnet_id, summary_registry_version)?
    .unwrap_or(EcdsaConfig {
        quadruples_to_create_in_advance: 1, // default value
        ..EcdsaConfig::default()
    });
```

Figure 3.1: `ic/rs/consensus/src/ecdsa/payload_builder.rs:400-405`

Signing requests are serviced by the system in the order in which they are made (as determined by their `CallbackID` values). If a canister (or set of canisters) makes a large number of signing requests, the system would be overwhelmed and would take a long time to recover.

This issue is partly mitigated by the fee that is charged for signing requests. However, we believe that the financial ramifications of this problem could outweigh the fees paid by attackers. For example, the type of denial-of-service attack described in this finding could be devastating for a DeFi application that is sensitive to small price fluctuations in the Bitcoin market.

Since the ECDSA threshold signature service is not yet deployed on the Internet Computer, it is unclear how the service will be used in practice, making the severity of this issue difficult to determine. Therefore, the severity of this issue is marked as undetermined.

Exploit Scenario

A malicious canister learns that another canister on the Internet Computer is about to request a time-sensitive signature on a message. The malicious canister immediately requests a large number of signatures from the signing service, exhausting the number of available quadruples and preventing the original signature from completing in a timely manner.

Recommendations

One possible mitigation is to increase the number of quadruples that the system creates in advance, making it more expensive for an attacker to carry out a denial-of-service attack on the ECDSA signing service. Another possibility is to run multiple signing services on multiple subnets of the Internet Computer. This would have the added benefit of protecting the system from resource exhaustion related to cross-network bandwidth limitations. However, both of these solutions scale only linearly with the number of added quadruples/subnets.

Another potential mitigation is to introduce a dynamic fee or stake based on the number of outstanding signing requests. In the case of a dynamic fee, the canister would pay a set number of tokens proportional to the number of outstanding signing requests whenever it requests a new signature from the service. In the case of a stake-based system, the canister would stake funds proportional to the number of outstanding requests but would recover those funds once the signing request completed.

As any signing service that depends on consensus will have limited throughput compared to a centralized service, this issue is difficult to mitigate completely. However, it is important that canister developers are aware of the limits of the implementation. Therefore, regardless of the mitigations imposed, we recommend that the DFINITY team clearly document the limits of the current implementation.

4. Aggregated signatures are dropped if their request IDs are not recognized

Severity: **Informational**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-DFTECDSA-4
Target: `ic/rs/consensus/src/ecdsa/payload_builder.rs`	

Description

The `update_signature_agreements` function populates the set of completed signatures in the ECDSA payload. The function aggregates the completed signatures from the ECDSA pool by calling `EcdsaSignatureBuilderImpl::get_completed_signatures`. However, if a signature's associated signing request ID is not in the set of ongoing signatures, `update_signature_agreements` simply drops the signature.

```
for (request_id, signature) in builder.get_completed_signatures(
    chain,
    ecdsa_pool.deref()
) {
    if payload.ongoing_signatures.remove(&request_id).is_none() {
        warn!(
            log,
            "ECDSA signing request {:?} is not found in
            payload but we have a signature for it",
            request_id
        );
    } else {
        payload
            .signature_agreements
            .insert(request_id, ecdsa::CompletedSignature::Unreported(signature));
    }
}
```

Figure 4.1: `ic/rs/consensus/src/ecdsa/payload_builder.rs:817-830`

Barring an implementation error, this should not happen under normal circumstances.

Recommendations

Short term, consider adding the signature to the set of completed signatures on the next ECDSA payload. This will ensure that all outstanding signing requests are completed.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup for the various automated analysis tools used during this audit.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

We ran Clippy on the `consensus` crate. This run did not generate any findings or code quality recommendations.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, simply run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be either a single rule, a directory of rules, or the name of a rule set hosted on the Semgrep registry.

We ran a number of custom Semgrep rules on the `consensus` crate. Since support for Rust is still experimental, we focused on locating the following small set of issues:

- The use of panicking functions like `assert`, `unreachable`, `unwrap`, and `expect` in production code (that is, outside unit tests)

```
rules:
  - id: panic-in-function-returning-result
    patterns:
      - pattern-inside: |
          fn $FUNC(...) -> Result<$T> {
              ...
          }
      - pattern-either:
          - pattern: $EXPR.unwrap()
          - pattern: $EXPR.expect(...)
    message: |
      `expect` or `unwrap` called in function returning a `Result`.
    languages: [rust]
    severity: WARNING
```

Figure C.1: `panic-in-function-returning-result.yaml`

```
rules:
- id: unwrap-outside-test
```

```
  patterns:
  - pattern: $RESULT.unwrap()
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.unwrap()
            ...
        }
    "
  message: Calling `unwrap` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.2: unwrap-outside-test.yaml

```
rules:
- id: expect-outside-test
  patterns:
  - pattern: $RESULT.expect(...)
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.expect(...)
            ...
        }
    "
  message: Calling `expect` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.3: expect-outside-test.yaml

- The use of the `as` keyword, which may silently truncate integers during casting (for example, casting `data.len()` to a u32, may truncate the input length on 64-bit systems)

```
rules:
- id: length-to-smaller-integer
  pattern-either:
  - pattern: $VAR.len() as u32
  - pattern: $VAR.len() as i32
  - pattern: $VAR.len() as u16
  - pattern: $VAR.len() as i16
  - pattern: $VAR.len() as u8
  - pattern: $VAR.len() as i8
  message: |
    Casting `usize` length to smaller integer size silently drops high bits
    on 64-bit platforms
```

```
  languages: [rust]
  severity: WARNING
```

Figure C.4: `length-to-smaller-integer.yaml`

- Unexpected comparisons before subtractions (for example, ensuring that x < y before subtracting y from x), which may indicate errors in the code

```
rules:
- id: switched-underflow-guard
  pattern-either:
    - patterns:
        - pattern-inside: |
                if $Y > $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y > $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y >= $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y >= $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y < $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y < $X {

                } else {
                    ...
                }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
                if $Y <= $X {
                    ...
```

```
                }
        - pattern-not-inside: |
                if $Y <= $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y > $X {

                } else {
                    ...
                }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
                if $Y >= $X {

                } else {
                    ...
                }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
                if $Y < $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y <= $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern: |
                if $X < $Y {
                }
                ...

  message: Potentially switched comparison in if-statement condition
  languages: [rust]
  severity: WARNING
```

Figure C.5: `switched-underflow-guard.yaml`

This run did not result in any issues or code quality recommendations.

Tarpaulin

Tarpaulin can be installed using Cargo by running `cargo install cargo-tarpaulin`. To execute Tarpaulin on Linux, simply run the command `cargo tarpaulin` in the crate root directory.

Note that we failed to obtain unit test coverage data using Tarpaulin because of a compilation issue. We also tried to use `llvm-cov` and `kcov` to generate coverage data but were unsuccessful.

D. Code Quality Recommendations

The following is a list of findings that were not identified as immediate security issues but may warrant further investigation.

ECDSA Consensus Integration

- The `update_quadruples_in_creation` function (which advances quadruple creation) is called after `update_ongoing_signatures` (which matches new signing requests against available quadruples) when a new ECDSA data payload is created in `create_data_payload` (in `src/ecdsa/payload_builder.rs`). This means that new quadruples will not be available until the next block. Reordering the calls would allow new signing requests to complete one block earlier.

```
update_ongoing_signatures(
    new_signing_requests,
    current_key_transcript,
    &mut ecdsa_payload,
    log.clone(),
)?;

// ... <redacted>

let mut new_transcripts = update_quadruples_in_creation(
    current_key_transcript,
    &mut ecdsa_payload,
    &mut transcript_cache,
    height,
    &log,
)?;
```

Figure D.1: New quadruples are not completed when signatures are matched against available quadruples in `create_data_payload`.

Bitcoin Network Integration

- The code occasionally uses Rust's `unwrap` function instead of `expect`. While these uses of `unwrap` should not result in a crash, we recommend replacing them with `expect`, which would help developers more quickly troubleshoot the issue in the unlikely event of a crash.

- Some features of the Bitcoin canister are not implemented. Most importantly, the canister does not communicate to the adapter that a transaction must be posted to the Bitcoin network.

- Because the replica running the Bitcoin canister and the one running the adapter are two different versions of software, they could grow out of sync on a node. Some

safety measures should be put in place to prevent the use of the Bitcoin feature when this occurs.