2PC-MPC in Rust: Initial Functional Correctness and Security Audit

DW-01

Prepared for dWallet Labs

Dr. Nadim Kobeissi
Erik Takke

Symbolic Software
In Collaboration with 3MI Labs

June 4, 2024

Abstract

dWallet Labs's *2PC-MPC* high-level Rust crate represents the practical software implementation of the *"2PC-MPC: Emulating Two Party ECDSA in Large-Scale MPC"* protocol, which introduces a novel cryptographic structure for enabling a non-collusive and UC-secure two-party Elliptic Curve Digital Signature Algorithm (ECDSA) scheme. This architecture is novel in that it allows one party to be fully centralized, while fully abstracting away the decentralization of the second party, allowing it to scale to an arbitrary n virtual parties.

This audit focused on evaluating the functional correctness and security of three main components: distributed key generation, presign, and sign protocols within a massively decentralized network context.

Symbolic Software's review finds supporting evidence highlighting the codebase's strengths in supporting scalability, security, and efficient communication, with linear communication scaling and near-constant computational complexity per party. The audit also identified some relatively minor areas requiring attention. Some were documented (such as the incomplete implementation of locality due to current limitations in bulletproofs), while others related to the discovery of a nonce reuse attack caused by the Rust code deviating from spec.

The audit further revealed a high degree of complexity within the Rust codebase, particularly at the protocol implementation level. Efforts have been made to mitigate these complexities by aligning the code comments with the paper's specifications. Despite these efforts, the complexity inherent in the code still poses significant challenges, underscoring the importance of thorough documentation and perhaps suggesting a need for simplifying the codebase to improve auditability and reduce the potential for errors.

Recommendations for future improvements, along with identified vulnerabilities and their potential mitigations, are provided to guide the next stages of development and deployment.

Contents

List of Acronyms

AHE	Additively Homomorphic Encryption	48
DKG	Distributed Key Generation	3
ECDSA	Elliptic Curve Digital Signature Algorithm	ii
EncDH	Encrypted Diffie-Hellman	27
EncDL	Encrypted Discrete Logarithm	27
MPC	Multi-Party Computation	10

1

Executive Summary

For the busy executive who just doesn't have the time!

1.1 | About This Audit

Ensuring secure transactions on the internet, especially in blockchain technology, is an area of continuous research. As blockchain technologies have evolved, so have the need for more robust security solutions, including through threshold cryptography. Threshold cryptography enhances security by splitting control over a private key among multiple participants, requiring a subset (threshold) of them to agree before any operation, like creating a digital signature, is executed.

However, traditional implementations of threshold ECDSA [1] suffer from two main drawbacks: high message complexity and the need for secure, direct communication channels between all participants, which becomes impractical as the number of participants increases. These challenges lead to inefficiencies, especially in large-scale networks, hindering true decentralization and the broad adoption of blockchain technologies.

The main target of this audit engagement introduces a novel approach called "2PC-MPC", which stands for two-party computation emulated within a multi-party computation framework. The value of this approach lies in how it scales security without introducing complexity for the end users, or clients. 2PC-MPC aims to achieve this while allowing one signatory party to be fully centralized and *"abstracting away"* the decentralization of the second party, effectively making it a virtually decentralized party whose internal decentralization structure can be rearranged at will and without major changes to the protocol as it is deployed.

In addition, unlike traditional models that require complex and secure direct communications among all participants, 2PC-MPC uses a broadcast channel. This reduces overhead because every participant sends and receives messages through a common channel without needing direct connections to others.

The main target of this audit is the Rust implementation of the 2PC-MPC protocol, which is designed to be used in a blockchain context. The protocol is implemented

as a Rust crate, which is a package of reusable code that can be shared with other developers.

The underlying crates used in the implementation of the 2PC-MPC protocol are also of interest. These crates provide the cryptographic primitives and utilities necessary for the protocol to function. The audit will focus on the security of these crates as well as the 2PC-MPC protocol itself.

This audit is a collaboration between Symbolic Software and 3MI Labs. Symbolic Software is a applied cryptography and security software auditing firm that specializes in security audits and cryptographic protocol analysis. 3MI Labs[1] is a research company that focuses on blockchain technologies and decentralized systems.

1.2 | What Was Audited

The audit focused on the following components:

- **The 2PC-MPC High-Level Rust Crate**: This implements distributed key generation, presigning, and signing protocols for enabling non-collusive and universally composable (UC-secure) two-party ECDSA in a massively decentralized network setting. The audit assesses the correctness and security of this core implementation.
- **The Accompanying Paper and Formal Specification**: The audit verifies that the Rust crate implementation aligns with the protocols described in the accompanying academic paper.
- **The Underlying Rust Crates**: Several lower-level crates providing cryptographic primitives like group operations, homomorphic encryption, zero-knowledge proofs, and commitments are reviewed for security and correctness.

The audit focuses on two core assessments: functional correctness (verifying the implementation matches specifications) and security (identifying potential vulnerabilities).

Due to time constraints, the audit had a limited scope, so further in-depth reviews are recommended, especially for the more complex underlying crates. Chapter 2 outlines the coverage level rating system used to gauge the depth of review for each component.

[1] Learn more about 3MI Labs by visiting https://www.3milabs.tech.

1.3 | About the Functional Correctness Assessment

Chapter 3 assesses the functional correctness of the three main subprotocols in 2PC-MPC: the Distributed Key Generation (DKG) subprotocol, the presign subprotocol, and the sign subprotocol. The assessment is based on the provided Rust implementation of the protocol, focusing on the cryptographic operations and protocol steps as described in the 2PC-MPC paper.

With the DKG subprotocol:

- The implementation follows Protocol 4 in the paper, using Rust's generic type system for the cryptographic primitives.
- It covers the centralized party's public key share commitment, decentralized party's initialization and encryption of secret key share, preparation for decentralized proof verification, and decommitment and verification of public key shares.
- The code aligns with the protocol specifications, ensuring secure generation and handling of secret and public key shares.

With the presign subprotocol:

- The implementation follows Protocol 5, preparing and ensuring the integrity and secrecy of nonce shares for distributed signature generation.
- It covers the centralized party's preparation (nonce generation, commitment, and proof construction), decentralized party's nonce and mask generation and encryption, proof aggregation initialization, and centralized party's verification of presign output.
- The code ensures secure handling of nonces, masks, and their associated proofs, aligning with the protocol specifications.

With the sign subprotocol:

- The implementation follows Protocol 6, generating a decentralized digital signature.
- It covers the centralized party's signature and proof setup (homomorphic encryption and zero-knowledge proofs), decentralized party's proof verification and signature setup (partial decryption and threshold decryption rounds).
- The code ensures secure computation and verification of signature components, adhering to the protocol specifications.

The assessment notes that while the implementation is functionally correct, it was conducted on a best-effort basis due to the high complexity of the target, and further analysis is encouraged. It also highlights that the implementation may differ in ordering from the protocol steps in the paper for optimization purposes.

1.4 | About the Security Assessment

Chapter 4 presents a security assessment of the provided Rust implementation of the 2PC-MPC protocol. The assessment focuses on identifying potential vulnerabilities, weaknesses, and deviations from best practices that could compromise the security of the cryptographic operations and the overall protocol.

The security assessment is based on a review of the codebase, with particular attention given to the handling of sensitive data, the implementation of cryptographic primitives, and the adherence to secure coding practices. The assessment aims to uncover any security issues that could lead to unauthorized access, data leakage, or the compromise of the protocol's integrity.

The chapter is structured around three main security findings, each presented in detail using the provided audit finding template. The findings are categorized based on their severity, ranging from low to critical, indicating the potential impact they could have on the security of the 2PC-MPC protocol:

1. **DW-01-001** (§4), identified as a critical issue, discusses the reuse of nonces in the decentralized party presigning step. Nonce reuse is a well-known vulnerability in cryptographic protocols that can lead to serious attacks, such as the recovery of private keys or the ability to forge signatures. The assessment provides recommendations for revising the nonce handling procedures to ensure the uniqueness and unpredictability of nonces used in the protocol.

2. **DW-01-002** (§4), classified as a medium severity issue, highlights insufficient checks on the `ComputationalSecuritySizedNumber` type. The current implementation of this type lacks proper validation and restrictions, potentially allowing values with insufficient entropy to be used in security-sensitive contexts. The assessment recommends implementing strict compile-time and runtime checks to enforce the required entropy criteria for this type.

3. **DW-01-003** (§4), rated as a low severity issue, addresses the lack of zeroization of secrets in memory. The 2PC-MPC crate does not currently employ any mechanisms to securely erase sensitive data from memory once it is no longer needed. This lack of zeroization could potentially leave secrets vulnerable to unauthorized access or exploitation. The assessment recommends utilizing the Rust `zeroize` crate to ensure the secure erasure of sensitive data from memory.

For each finding, the assessment provides detailed explanations of the issue, including relevant code snippets and the potential impact on the security of the protocol. Additionally, the assessment offers specific recommendations and mitigation strategies to address the identified vulnerabilities and strengthen the overall security posture of the 2PC-MPC implementation.

1.5 | Summary of Conclusions

Chapter 5 presents a comprehensive summary of the analysis and assessment conducted on the Rust implementation of the 2PC-MPC protocol.

§ 5.1 recaps the key findings from the functional correctness assessment and the security assessment. The functional correctness assessment, which compared the implementation against the protocol specifications, found that the Rust code generally aligns with the protocol descriptions. However, further analysis is recommended due to the high complexity of the target. The security assessment identified three main security findings of varying severity levels, highlighting issues such as nonce reuse, insufficient checks on security-sensitive types, and lack of zeroization of secrets in memory.

§ 5.2 discusses the exceptionally high complexity of the Rust implementation, which can make the codebase challenging to understand, maintain, and audit. The chapter provides recommendations for future development to address the complexity issue, including code simplification, comprehensive documentation, rigorous testing, regular security audits and community engagement.

§ 5.3 proposes several avenues for further exploration and development to enhance the protocol's security, efficiency, and usability. These include an in-depth cryptographic review, cryptographic optimizations, protocol API correctness and usability analysis, state machine transition analysis, understanding the decentralized party in a protocol setting, performance and scalability testing, integration with existing systems and frameworks, and real-world applications and case studies.

The chapter emphasizes the importance of conducting an in-depth protocol analysis that treats each virtually decentralized party as a distinct individual party, considering factors such as party composition, communication patterns, key management, fault tolerance, scalability, privacy, and security.

2

Target Overview

This chapter describes the three main targets of this audit:

1. **2PC-MPC High-Level Rust Crate.** The main target of the audit, implementing the full distributed key generation, presign, and sign subprotocols as described in the accompanying paper/formal specification. It is designed to offer a two-party ECDSA scheme that is not only non-collusive and UC-secure but also capable of handling the demands of a massively decentralized network with scalability in both communication and computation. Our goal here is to assess the correctness and security of the implementation, ensuring that it aligns with the theoretical specifications and claims of the protocol.
2. **Accompanying Paper and Formal Specification.** The audit assesses whether the implementation in the Rust crate aligns with the protocol as described in the accompanying academic paper. This includes a verification of the protocol's claims such as UC security, public verifiability, and the efficiency of its communication and computational complexities. The fidelity of the crate to the paper's specifications helps assess whether the theoretical security properties hold in the practical deployment.
3. **Underlying Rust Crates.** The functionality of the 2PC-MPC crate relies on several lower-level crates, each responsible for different cryptographic primitives and operations. These include crates for handling group operations, homomorphic encryption, zero-knowledge proofs, and commitments. Our audit examines these crates for security and correctness, ensuring that they robustly support the higher-level operations of the 2PC-MPC crate.

The audit's assessment guidelines focus on two core assessments:

1. **Functional Correctness Assessment.** The audit verifies that the implementation of the protocol in the 2PC-MPC crate aligns with the specifications outlined in the accompanying paper. This includes ensuring that the cryptographic operations are correctly implemented, the protocol's communication and computational complexities are as claimed, and the protocol's security properties are maintained.

2. **Security Assessment.** The audit evaluates the security of the 2PC-MPC crate and its underlying crates, examining them for potential vulnerabilities that could compromise the security of the protocol. This includes assessing the crates for common cryptographic vulnerabilities, ensuring that they follow best practices in secure implementation, and evaluating their resistance to potential attacks.

The inherent complexities involved, the limited timeframe for the engagement and the breadth of the codebase restricted the depth of review possible for each component. Consequently, certain areas, particularly some of the more intricate aspects of the underlying crates and their integration with the main *2PC-MPC* crate, did not undergo an exhaustive examination. It is highly recommended that further in-depth reviews be undertaken to ensure a comprehensive evaluation of all components. Additional scrutiny will likely yield further insights and potentially uncover subtleties not detected in this initial audit, thereby strengthening the overall security and robustness of the project.

The *"Coverage Level"* rating system used in this report is intended only as an intuitive general guideline intended to provide a high-level overview of the audit's scope and depth. It should not be interpreted as a definitive measure of the audit's quality or completeness.

2.1 | *2PC-MPC* Crate

Coverage Level: ●●●○○

- **Repository:** `https://github.com/dwallet-labs/2pc-mpc`
- **Branch:** `main`[1]
- **Commit:** `5891305042f8b13810b8d1d4e15cec5e4e6efea6`

This main crate integrates distributed key generation, presign, and sign protocols essential for multiparty ECDSA. It aims to provide a non-collusive and UC-secure two-party ECDSA scheme that scales linearly in communication while maintaining near-constant computational complexity.

2.1.1 | Main Subprotocols

The audit focused on assessing the functional correctness and security of each of the three main subprotocols:

[1]When initially audited, the same target was offered under a different branch, `instantiations`, with the commit hash `057d0420dc2149543e417f7689f5335683df14c6`. This same target has since been merged into the `main` branch, under the commit hash listed above.

2.1.1.1 | Distributed Key Generation

DKG is designed to establish the cryptographic framework necessary for subsequent secure communications and transactions. Initially, party A generates a random private key, x_A, from which it computes the corresponding public key, X_A. This public key is then broadcasted to all participants for verification. Simultaneously, each participant B_i generates its own private key x_i and computes the respective public key X_i. Each B_i then encrypts its private key to collectively create $\mathtt{ct}_{\mathtt{key}}$, which is a core part of ensuring that the collective public key X_B can be used without exposing individual private keys. This phase concludes with all parties verifying the integrity and correctness of the public keys involved; any discrepancy or failure in validation leads to aborting the process and identifying the responsible parties.

2.1.1.2 | Presign

The presigning phase is critical for setting up secure and verifiable transactions. Here, party A starts by generating a random nonce k_A. Following this, each participant B_i contributes by generating their own nonces and encrypted values that collectively will form the secure environment for the transaction. These include generating random values k_i, computing corresponding public values R_i, and encrypting these values alongside additional parameters designed to mask and secure the transaction process. Each B_i's contributions are verified through a series of cryptographic proofs to ensure that all values are correctly generated and committed. The presigning phase concludes with the generation of the final signature shares, which are then broadcasted for further verification.

2.1.1.3 | Sign

Signing completes the process by creating a valid ECDSA signature that can be used to authenticate a transaction securely. Party A computes a preliminary signature component using its private keys, the previously committed values, and the encrypted information shared by other participants during the presigning phase. The decentralized participants B_i then collectively verify A's computations and contribute their respective parts of the signature by decrypting and adjusting the signature share using a shared random factor γ. The final signature, composed of r and s, is calculated using a combination of all shared contributions, ensuring that the signature is both valid and verifiable.

2.1.2 | Understanding the Decentralized Party in a Protocol Setting

In this audit report, we have primarily focused on the interaction between two main entities: the centralized party and the decentralized party. The centralized party is treated as a single, unified entity, while the decentralized party is considered as a single, virtually merged entity representing multiple participants. However, it is important

to acknowledge that this simplification may not fully capture the nuances and complexities of the type of truly decentralized system that 2PC-MPC aims to facilitate.

In reality, the decentralized party consists of multiple individual parties, each with their own unique characteristics, behaviors, and potential for change. These parties may join or leave the system dynamically, merge with other parties, or undergo internal changes that affect their roles and responsibilities within the protocol.

To gain a more comprehensive understanding of the 2PC-MPC protocol and its implementation, future work should include an in-depth protocol analysis that treats each virtually decentralized party as a distinct individual party. This analysis should take into account the following considerations:

- **Party Composition**: Examine the composition of the decentralized party, considering the number of individual parties involved, their roles, and their relationships with each other. Analyze how the protocol handles the addition of new parties, the removal of existing parties, and the potential for parties to merge or split.
- **Communication Patterns**: Investigate the communication patterns among the individual parties within the decentralized party. Analyze how messages are exchanged, how consensus is reached, and how potential conflicts or disagreements are resolved. Consider the impact of network latency, asynchronous communication, and the possibility of malicious or faulty parties.
- **Key Management**: Explore the key management aspects of the protocol, focusing on how cryptographic keys are generated, distributed, and managed among the individual parties. Analyze the security implications of key sharing, key rotation, and the potential for key compromise or leakage.
- **Fault Tolerance**: Assess the protocol's resilience to failures or misbehavior of individual parties within the decentralized party. Examine how the protocol handles scenarios such as party disconnections, network partitions, or malicious actions. Evaluate the protocol's ability to maintain consistency, integrity, and availability in the presence of faults.
- **Scalability**: Investigate the scalability aspects of the protocol, considering how it performs as the number of individual parties within the decentralized party increases. Analyze the impact of larger party counts on communication overhead, computational complexity, and overall system performance.
- **Privacy and Security**: Assess the privacy and security guarantees provided by the protocol when treating each virtually decentralized party as an individual party. Analyze the potential for information leakage, collusion, or attacks that may arise from the interactions among individual parties. Evaluate the effectiveness of the protocol's privacy-preserving mechanisms and the robustness of its security properties.

By conducting an in-depth protocol analysis that considers the decentralized party as a collection of individual parties, we can gain a more comprehensive understanding of the 2PC-MPC protocol's behavior, security, and performance in a truly decentralized setting. This analysis will help identify potential challenges, vulnerabilities, and areas for improvement that may arise from the complexities of decentralized party composition and interaction.

Furthermore, this analysis can provide valuable insights into the protocol's adaptability and resilience to changes in the decentralized party landscape. By examining how the protocol handles party additions, mergers, and shifts, we can assess its ability to accommodate the dynamic nature of decentralized systems and ensure its continued effectiveness and security.

The insights gained from this decentralized party composition analysis will contribute to the ongoing development and refinement of the 2PC-MPC protocol and its implementation. By addressing the nuances and challenges associated with treating virtually decentralized parties as individual parties, we can enhance the protocol's robustness, scalability, and practical applicability in real-world decentralized environments.

2.2 | *2PC-MPC* Accompanying Paper

Coverage Level: ●●●○○

The *2PC-MPC* high-level Rust crate is paired with a formal specification, currently available as an ePrint draft, *"2PC-MPC: Emulating Two Party ECDSA in Large-Scale MPC"* [2], which serves as a foundational specification for the crate, detailing an original cryptographic protocol designed for massively decentralized networks. The protocol enables the emulation of two-party ECDSA signing in a Multi-Party Computation (MPC) setting, where one party A is fully centralized, while party B is virtually decentralized such that its decentralization can be *"abstracted away"*, and consists of a threshold of n participants. The protocol is designed to be universally composable (UC-secure) and publicly verifiable.

The construction appears to be largely inspired by previous threshold ECDSA work (for example [1]), with the primary difference being the use of a two-party MPC protocol to emulate the threshold ECDSA protocol. To accomplish this, the paper employs a novel combination of Paillier encryption [3], Maurer's universal zero-knowledge proofs [4], bullet proofs [5] and Pedersen commitments [6] to ensure the security and privacy of the protocol.

The paper makes several key claims regarding the protocol and its implementation:

1. **UC-Secure and Publicly Verifiable Protocols.** The paper introduces threshold ECDSA protocols that are meant to be universally composable (UC-secure) but also publicly verifiable. This ensures that the protocols can be safely com-

posed with other cryptographic protocols without compromising overall security, and the outcomes of protocol executions are verifiable by any external observer.

2. **Reduction in Complexity.** A significant achievement detailed in the paper is the reduction of message complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ and computational complexity from $\mathcal{O}(n)$ to practically $\mathcal{O}(1)$ per participant, where n is the number of parties.

3. **Broadcast Communication Channel.** Unlike traditional multiparty computation protocols that may require peer-to-peer communication, this protocol operates exclusively over a broadcast channel. This design choice is particularly suited to applications such as permissionless blockchain bridges and decentralized custody solutions, where establishing direct channels between all parties is unfeasible.

2.3 | Underlying Crates

The audit targets several Rust crates that are integral to the implementation of the "2PC-MPC: Emulating Two Party ECDSA in Large-Scale MPC" protocol, each contributing distinct cryptographic functionality.

The underlying crates were audited according to the version imported by the *2PC-MPC* crate as of the commit repository, hash and branch mentioned in §2.1. The crates audited include:

- **`group`**. Defines traits for abelian groups to ensure they operate securely under cryptographic protocols, supporting both dynamic and static groups. Its structure allows for the use of specific group types in higher-level schemes and ensures that group operations are constant-time and secure against malicious inputs. *Coverage Level:* ●●●●○

- **`homomorphic-encryption`**. Establishes traits for homomorphic encryption, including threshold schemes, which are crucial for maintaining data privacy while allowing computations on encrypted data. *Coverage Level:* ●●●○○

- **`proof`**. Provides traits and helpers for constructing zero-knowledge proofs and range proofs, facilitating the verification of claims without revealing underlying data. *Coverage Level:* ●●●●○

- **`commitment`**. Offers traits for creating homomorphic and non-homomorphic commitment schemes, including implementations based on Pedersen commitments and hash-based techniques using `merlin::Transcript`. *Coverage Level:* ●●●●○

- **`maurer`** and **`enhanced-maurer`**. These crates implement generic Maurer zero-knowledge proofs [4] that can be adapted for various group homomorphisms, with the enhanced version including additional capabilities for range claims. *Coverage Level:* ●●○○○

Prior to the commissioning of this audit, most of the above crates had been subject to a self-reported internal auditing process that includes review by cryptographers and programmers. However, none of these components had undergone a third-party audit. The audit has thus focused on verifying their adherence to the cryptographic specification and their functional correctness assessing their implementation complexity, and evaluating their security measures against potential vulnerabilities.

3

Functional Correctness Assessment

This chapter covers the assessment of functional correctness for 2PC-MPC's three main subprotocols: the DKG subprotocol, the presign subprotocol, and the sign subprotocol. The assessment is based on the provided Rust implementation of the protocol, focusing on the cryptographic operations and the protocol steps as described in the 2PC-MPC paper. [2]

In assessing functional correctness, we examine the implementation of the cryptographic operations and the protocol steps to ensure that the code aligns with the protocol specifications. We also evaluate the code for potential vulnerabilities, ensuring that the cryptographic operations are secure and that the protocol steps are correctly implemented.[1]

The assessment is structured around the protocol steps outlined in the 2PC-MPC paper, focusing on the key cryptographic operations and the protocol steps that are essential for the secure multi-party computation of digital signatures. The assessment is based on the provided Rust code, which encapsulates the cryptographic operations and the protocol steps in a modular and reusable manner.

It is important to note that this functional correctness assessment was conducted on a best-effort basis, given the unusually high complexity of the target. As such, further analysis is encouraged.

3.1 | DKG Subprotocol

Protocol 4 (Figure A.1) in the *"2PC-MPC: Emulating Two Party ECDSA in Large-Scale MPC"* paper describes the key generation step for setting up a secure multi-party computation environment. This implementation is done in Rust, where the Rust generic type system is exploited in order to encapsulate the high-level cryptographic operations.

The Rust code defines a generic struct named `Party` parameterized over various constants and types that correspond to the cryptographic primitives used in the protocol.

[1] Security findings are covered in Chapter 4.

The struct includes public parameters for the group, encryption scheme, and zero-knowledge proof systems.

```
  pub struct Party<
    const SCALAR_LIMBS: usize,
    const COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS: usize,
    const RANGE_CLAIMS_PER_SCALAR: usize,
    const PLAINTEXT_SPACE_SCALAR_LIMBS: usize,
    GroupElement: PrimeGroupElement<SCALAR_LIMBS>,
    EncryptionKey: AdditivelyHomomorphicEncryptionKey<
    ↪ PLAINTEXT_SPACE_SCALAR_LIMBS>,
    RangeProof: AggregatableRangeProof<
    ↪ COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS>,
    UnboundedEncDLWitness: group::GroupElement + Samplable,
    ProtocolContext: Clone + Serialize,
> {
    protocol_context: ProtocolContext,
    scalar_group_public_parameters: group::PublicParameters<GroupElement
    ↪ ::Scalar>,
    group_public_parameters: GroupElement::PublicParameters,
    encryption_scheme_public_parameters: EncryptionKey::PublicParameters,
    unbounded_encdl_witness_public_parameters: UnboundedEncDLWitness::
    ↪ PublicParameters,
    range_proof_public_parameters: RangeProof::PublicParameters<
    ↪ RANGE_CLAIMS_PER_SCALAR>,
}
```

3.1.1 | Centralized Party Public Key Share Commitment

Relevant Code:

DIR `src/dkg/centralized_party/commitment_round.rs`

The function `sample_commit_and_prove_secret_key_share` executes the initial step of Protocol 4, responsible for securely generating a secret key share and its corresponding public commitment.

3.1.1.1 | Sampling the Secret Key Share

In compliance with Step 1a of Protocol 4, the participant samples a random scalar x_A from the scalar group, represented by `GroupElement::Scalar`. This scalar acts as the secret key share of the participant.

```
let secret_key_share = GroupElement::Scalar::sample(
    &self.scalar_group_public_parameters, rng)?;
```

3.1.1.2 | Public Key Share Derivation

The public key share $X_A = x_A \cdot G$ is derived from the sampled secret key share by multiplying it with the group's generator G. This operation converts the secret scalar into a point on the elliptic curve, thereby forming the public key share.

```
let public_key_share: GroupElement = public_key_share
    .first()
    .ok_or(crate::Error::InternalError)?
    .clone();
```

3.1.1.3 | Commitment to the Public Key Share

Following the generation of the public key share, the participant creates a commitment to this public key. This commitment is facilitated by a cryptographic function that securely binds the public key with randomly chosen commitment randomness. This action aligns with Step 1b of Protocol 4, ensuring the commitment is ready for the zero-knowledge proof.

```
let commitment = commit_public_key_share(
    CENTRALIZED_PARTY_ID,
    &public_key_share,
    &commitment_randomness,
)?;
```

3.1.1.4 | Generation of Zero-Knowledge Proof

The participant generates a zero-knowledge proof that they know the discrete logarithm of the public key share relative to the base G. This proof is essential for verifying that the participant possesses the corresponding secret key of the publicly committed key share without revealing the secret key itself. This proof generation corresponds to Step 1b, where the commitment and the knowledge of the secret key are provably linked.

```
let (knowledge_of_discrete_log_proof, _) = knowledge_of_discrete_log::
    ↪ Proof::<
    GroupElement::Scalar,
    GroupElement,
    ProtocolContext,
>::prove(
    &self.protocol_context,
    &language_public_parameters,
    vec![secret_key_share],
    rng,
)?;
```

After generating the commitment and the zero-knowledge proof, the function encapsulates all relevant data into a structure for the next steps of the protocol. This structure includes the secret key share, public key share, the proof of knowledge, and the commitment randomness, preparing the participant for any decentralized verification processes that follow.

```
let party = decommitment_round::Party {
    ...
    secret_key_share,
    public_key_share,
    knowledge_of_discrete_log_proof,
    commitment_randomness,
};
```

3.1.1.5 | Centralized Party Public Key Share Commitment

The function `commit_public_key_share` encapsulates the process of committing to a public key share during the protocol, creating a verifiable commitment that can be checked by other parties in the protocol, ensuring that the public key share is not tampered with.

```
pub fn commit_public_key_share<GroupElement: group::GroupElement>(
  party_id: PartyID,
  public_key_share: &GroupElement,
  commitment_randomness: &ComputationalSecuritySizedNumber,
) -> crate::Result<Commitment> {
  let mut transcript = Transcript::new(b"DKG commitment round of
    ↪ centralized party");

  transcript
    .serialize_to_transcript_as_json(b"public key share", &
    ↪ public_key_share.value())
    .unwrap();

  Ok(Commitment::commit_transcript(
    party_id,
    "DKG commitment round of centralized party".to_string(),
    &mut transcript,
    commitment_randomness,
  ))
}
```

This function utilizes a transcript for recording cryptographic operations, which is a common technique in modern cryptographic protocols to ensure that the operations can be audited and replayed for verification. Under the hood, the `merlin` [7] Rust composable proof transcripts library is used to manage the transcript state and serialize the data for commitment.

3.1.2 | Initialization and Encryption of Secret Key Share

Relevant Code:

DIR src/dkg/decentralized_party/encryption_of_secret_key_share_round.rs

The Rust code function `sample_secret_key_share_and_initialize_proof_aggregation` encapsulates the steps necessary for sampling the secret key share and preparing it for further cryptographic operations:

- **Sampling Secret Key Share x_i:** The secret key share is sampled from a predefined scalar group, ensuring that the keys are cryptographically secure. This is implemented as follows:

```
let share_of_decentralized_party_secret_key_share = GroupElement::
    ↪ Scalar::sample(&self.scalar_group_public_parameters, rng)?;
```

  This code snippet directly corresponds to Step 2b of Protocol 4 in the paper, where each decentralized party samples their secret key share.

- **Sampling Randomness ρ_i:** Alongside the key share, a corresponding randomness value is also sampled to be used in the encryption process:

```
let encryption_randomness = EncryptionKey::
    ↪ RandomnessSpaceGroupElement::sample(
    &self.encryption_scheme_public_parameters
        .as_ref()
        .randomness_space_public_parameters, rng)?;
```

  This step ensures the unpredictability of the encrypted values, providing additional security, which reflects the Step 2d of Protocol 4.

- **Encryption and Language Enhancement:** The sampled key share and the randomness are used to initialize the enhanced public parameter struct for encryption:

```
let language_public_parameters = EnhancedPublicParameters::new(
    self.unbounded_encdl_witness_public_parameters.clone(),
    self.range_proof_public_parameters.clone(),
    language_public_parameters,
)?;
```

  This setup integrates enhanced cryptographic protocols (e.g., range proofs) to ensure the robustness and flexibility of the encryption scheme.

3.1.3 | Preparation for Decentralized Proof Verification

Relevant Code:

DIR src/dkg/decentralized_party/encryption_of_secret_key_share_round.rs

The encrypted key share and its randomness are then prepared for decentralized proof verification, facilitating collective validation:

```
  let share_of_decentralized_party_secret_key_share_witness =
    ↪ EnhancedLanguage::<
  SOUND_PROOFS_REPETITIONS,
  RANGE_CLAIMS_PER_SCALAR,
  COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS,
  RangeProof,
  UnboundedEncDLWitness,
  encryption_of_discrete_log::Language<
    PLAINTEXT_SPACE_SCALAR_LIMBS,
    SCALAR_LIMBS,
    GroupElement,
    EncryptionKey,
  >,
>::generate_witness(
  (
    EncryptionKey::PlaintextSpaceGroupElement::new(
      Uint::<PLAINTEXT_SPACE_SCALAR_LIMBS>::from(
        &share_of_decentralized_party_secret_key_share_value,
      )
      .into(),
      self.encryption_scheme_public_parameters
        .plaintext_space_public_parameters(),
    )?,
    encryption_randomness,
  )
    .into(),
  &language_public_parameters,
  rng,
)?;
```

This snippet illustrates how the cryptographic elements are readied for the next stage where they will be collectively verified. The code prepares the witness for the decentralized proof verification, ensuring that the encrypted key share is correctly encrypted and can be validated by other parties in the protocol.

3.1.4 | Decommitment and Verification of Public Key Shares

Relevant Code:

DIR src/dkg/centralized_party/decommitment_round.rs

This section of the cryptographic process deals with the decommitment and verification of the cryptographic proofs related to the public key shares. The provided Rust

implementation validates commitments and ensures the correct reconstruction of shared public keys.

The function `decommit_proof_public_key_share` encapsulates the steps necessary for decommitting and verifying the public key shares:

- **Reconstruction of Encrypted Keys:** The function begins by reconstructing the encrypted key shares, verifying that the encryption adheres to the expected parameters.

  ```
  let encrypted_decentralized_party_secret_key_share =
      EncryptionKey::CiphertextSpaceGroupElement::new(...);
  ```

  This step ensures that the encrypted keys are reconstructed correctly from the provided proofs and commitments.

- **Verification of Public Key Shares:** The public key share from the decentralized party is then reconstructed:

  ```
  let decentralized_party_public_key_share = GroupElement::new(...);
  ```

  This step confirms the authenticity of the public key share, ensuring it matches the one originally committed to, aligning with step 3 of Protocol 4.

- **Verification of Commitment and Proof:** The function proceeds to verify the commitments and the cryptographic proofs associated with the public key shares:

  ```
  decentralized_party_secret_key_share_encryption_and_proof
      .encryption_of_secret_key_share_proof.verify(...);
  ```

  This proof verification ensures that the shared keys and their commitments have been handled securely and according to the protocol specifications, reflecting thorough integrity checks.

After successful verification, the function computes the combined public key and prepares outputs and state transitions for further cryptographic operations:

```
let public_key = self.public_key_share.clone() + &
    ↪ decentralized_party_public_key_share;
```

This step effectively combines the public key shares to form a unified public key, marking the completion of the decommitment and verification processes. The function also prepares output structures that encapsulate the results of this phase, facilitating transparency and traceability of the operations performed.

3.1.5 | Decentralized Party Decommitment Proof Verification Round

Relevant Code:

DIR src/dkg/decentralized_party/decommitment_proof_verification_round.rs

This part of the cryptographic process involves the verification of commitments and proofs relating to the public key shares. The Rust implementation validates the decommitment and checks the proofs of discrete logs associated with the public keys.

The Rust function `verify_decommitment_and_proof_of_centralized_party_public_key_share` encapsulates the verification steps:

- **Reconstructing and Verifying Commitment:** The function begins by reconstructing the commitment from the public key share disclosed by the centralized party and comparing it with the original commitment to ensure that it has not been tampered with:

```
let reconstructed_commitment = commit_public_key_share(
    CENTRALIZED_PARTY_ID, &centralized_party_public_key_share,
    &decommitment_and_proof.commitment_randomness);
```

  This step confirms the consistency of the value revealed by party A against the stored commitment, reflecting the integrity checks described in steps 4 and 5 of Protocol 4 in the paper.

- **Verification of Proof of Decommitted Value:** The cryptographic proof accompanying the decommitment is verified to ascertain that it is indeed valid and corresponds accurately to the shared public key element:

```
decommitment_and_proof.proof.verify(
    &self.protocol_context, &language_public_parameters,
    vec![centralized_party_public_key_share.clone()]);
```

  This verification ensures that the proof of knowledge for the discrete logarithm of the public key share is correct, as required by the protocol to maintain security.

- **Aggregate Public Key Construction:** Following successful verification, the public keys from both centralized and decentralized parties are aggregated to construct the combined public key:

```
let public_key = centralized_party_public_key_share.clone() + &
    ↪ public_key_share;
```

This step combines the public keys from both parties to form a unified public key, completing the setup for the cryptographic operations that follow, aligning with Step 4a in Protocol 4.

3.2 | Presign Subprotocol

Protocol 5 (Figure A.2) refers to 2PC-MPC's presign subprotocol, which deals with preparing and ensuring the integrity and secrecy of the nonce shares used for generating a digital signature in a distributed manner. It involves several steps, including the generation of nonces, the creation of cryptographic commitments, and the construction of zero-knowledge proofs to verify these commitments without revealing the underlying values.

The following sections summarize the functional correctness assessment of the presigning subprotocol as realized in the provided Rust code.

3.2.1 | Step 1: Centralized Party Preparation

Relevant Code:

DIR `src/presign/centralized_party/commitment_round.rs`

Step 1 of Protocol 5 (Figure A.2) involves the centralized party generating a nonce and creating a cryptographic commitment to it, followed by constructing a proof that they know the value committed to without revealing it. The provided Rust code implements these steps through the following mechanisms:

3.2.1.1 | Nonce Generation

The participant samples a random nonce k_A using the `sample_batch` method from the `GroupElement::Scalar` structure. This nonce serves as the participant's contribution to the protocol.

```
let signature_nonce_shares = GroupElement::Scalar::sample_batch(
    &self.scalar_group_public_parameters,
    batch_size,
    rng,
)?;
```

3.2.1.2 | Commitment Generation

A commitment to the nonce k_A is generated using a Pedersen commitment scheme. This is facilitated by the `sample_batch` method which samples a random scalar for the commitment randomness ρ_1.

```
let commitment_randomnesses = GroupElement::Scalar::sample_batch(
    &self.scalar_group_public_parameters,
    batch_size,
    rng,
)?;
```

The commitments are generated using the public parameters derived for the Pedersen commitment scheme. Each nonce and its corresponding randomness are paired and used to generate the commitment.

```
let commitments = signature_nonce_shares_and_commitment_randomnesses
    .iter()
    .map(|(nonce_share, commitment_randomness)| Pedersen::commit(
        nonce_share, commitment_randomness, &
    ↪ commitment_scheme_public_parameters
    ))
    .collect::<Vec<_>>();
```

3.2.1.3 | Proof Construction

Using the `maurer::Proof::prove` method, a zero-knowledge proof of knowledge for the decommitment of the nonce is constructed. This proof demonstrates that the participant knows the value k_A and the randomness ρ_1 used in the commitment without revealing them.

```
let (proof, _) = maurer::Proof::<...>::prove(
    &self.protocol_context,
    &language_public_parameters,
    signature_nonce_shares_and_commitment_randomnesses
        .clone()
        .into_iter()
        .map(|(nonce_share, commitment_randomness)| {
            ([nonce_share].into(), commitment_randomness).into()
        })
        .collect(),
    rng,
)?;
```

3.2.2 | Step 2a: Decentralized Party Nonce and Mask Generation

Relevant Code:

FILE `encrypted_masked_key_share_and_public_nonce_shares_round.rs`[2]

Step 2a of the protocol involves multiple participants collaboratively preparing to engage in the cryptographic operations that underpin the multi-party computation of digital signatures. Specifically, the operations involve sampling masks and nonce shares, and initiating the process for their secure aggregation and proof verification.

As discussed earlier in this chapter, the primary structure, `Party`, encompasses all necessary cryptographic parameters and methods to perform the operations required in Step 2.

[2] The file resides in the directory `src/presign/decentralized_party`.

3.2.2.1 | Verification of Commitments

This segment verifies the commitments made in the previous steps of the protocol. Each participant checks the integrity and correctness of the nonce shares committed by their peers. The verification uses a language and proof system defined within the protocol, ensuring that all parties hold valid commitments before proceeding.

```
let centralized_party_nonce_shares_commitments =
    centralized_party_nonce_shares_commitments_and_batched_proof
    .commitments
    .into_iter()
    .map(|value| GroupElement::new(value, &self.group_public_parameters))
    .collect::<group::Result<Vec<_>>>()?;
centralized_party_nonce_shares_commitments_and_batched_proof
    .proof
    .verify(
        &self.protocol_context,
        &l_dcom_public_parameters,
        centralized_party_nonce_shares_commitments.clone(),
    )?;
```

3.2.2.2 | Sampling of Masks and Nonce Shares

In this step, each participant samples their respective shares of the signature nonce (k_i). The nonces are then converted into a form suitable for encryption, using the public parameters of the encryption scheme, allowing them to be encrypted in subsequent steps:

```
let shares_of_signature_nonce_shares_witnesses = masks_shares
.clone()
.into_iter()
.map(|share_of_signature_nonce_share| {
  let share_of_signature_nonce_share_value: Uint<SCALAR_LIMBS> =
    share_of_signature_nonce_share.into();

  EncryptionKey::PlaintextSpaceGroupElement::new(
    Uint::<PLAINTEXT_SPACE_SCALAR_LIMBS>::from(
      &share_of_signature_nonce_share_value,
    )
    .into(),
    self.encryption_scheme_public_parameters
      .plaintext_space_public_parameters(),
  )
})
.collect::<group::Result<Vec<_>>>()?;
```

!

Nonce Reuse Vulnerability in Nonce Share Sampling

The implementation of the presigning subprotocol samples a batch of mask shares for generating nonces (referred to as γ_i) and then reuses the same randomness in the above code for a different cryptographic purpose, namely generating signature nonce shares (referred to as k_i). This results in a nonce reuse vulnerability, which can lead to the compromise of the cryptographic operations and the security of the protocol. The vulnerability is disclosed in detail in Chapter 4.

!

Subtle Differences in Multiplicative Group Usage

The 2PC-MPC paper's description of Protocol 5 (Figure A.2) states that in Step 1a, k_A should be sampled from $\mathbb{Z}_q$ and that in Step 2a, the k_i nonce shares should be sampled from $\mathbb{Z}_q^*$. However, the paper does not formally define $\mathbb{Z}_q^*$. dWallet Labs has clarified that $\mathbb{Z}_q^*$ is simply $\mathbb{Z}_q$ without the 0 element. dWallet Labs further emphasized that given the size of $\mathbb{Z}_q$, the probability of k_A or k_i being sampled as 0 is negligible, and that therefore either $\mathbb{Z}_q$ or $\mathbb{Z}_q^*$ may be used for sampling k_A and k_i.
In addition, the implementation will output an error if:

- The inversion of k_A by the centralized party in the first steps of Protocol 6 (Figure A.3) fails, which appears to only be possible if $k_A = 0 \bmod q$.
- The inversion of $\mathtt{pt}_4$ by a decentralized party in the final steps of Protocol 6 (Figure A.3) fails, which appears to only be possible if $\mathtt{sum}_i(k_i) = k_B = 0 \bmod q$.

Following the sampling of k_i, the protocol requires the generation of masks encryption randomness ($\eta^i_{mask_1}$), masked key share encryption randomness ($\eta^i_{mask_2}$) and encryption randomness ($\eta^i_{mask_3}$) for each nonce share. Each randomness variable serves a specific purpose within the cryptographic framework of the protocol:

- $\eta^i_{mask_1}$ (**`masks_encryption_randomness`**): Used to encrypt the mask shares (γ_i), securing the mask values for computation of $\mathtt{ct}^i_1$ and $\mathtt{ct}^i_2$. Sampled here:

```
let masks_encryption_randomness = EncryptionKey::
    ↪ RandomnessSpaceGroupElement::sample_batch(
    self.encryption_scheme_public_parameters
        .randomness_space_public_parameters(),
    batch_size,
    rng,
)?;
```

- $\eta^i_{mask_2}$ (**`masked_key_share_encryption_randomness`**): Employed to encrypt the masked key shares, which are computed by combining mask shares with a confidential key component. Sampled here:

```
let masked_key_share_encryption_randomness =
    EncryptionKey::RandomnessSpaceGroupElement::sample_batch(
        self.encryption_scheme_public_parameters
            .randomness_space_public_parameters(),
        batch_size,
        rng,
    )?;
```

- $\eta^i_{mask_3}$ (**`shares_of_signature_nonce_shares_encryption_randomness`**): Utilized for encrypting the nonce shares (k_i), critical for signature generation processes. Sampled here:

```
let shares_of_signature_nonce_shares_encryption_randomness =
    EncryptionKey::RandomnessSpaceGroupElement::sample_batch(
        &self
            .encryption_scheme_public_parameters
            .as_ref()
            .randomness_space_public_parameters,
        batch_size,
        rng,
    )?;
```

3.2.3 | Step 2a: Decentralized Party Mask Encryption

Relevant Code:

DIR `src/presign/decentralized_party.rs`

The randomness elements sampled above are crucial for maintaining confidentiality and integrity in the cryptographic operations:

- $\mathbf{ct}^i_1$ (**`encrypted_masks`**): Generated by encrypting γ_i using $\eta^i_{mask_1}$:

```
let encrypted_masks: Vec<_> = masks_and_encrypted_masked_key_share
    .iter()
    .map(|mask_and_encrypted_masked_key_share| {
        mask_and_encrypted_masked_key_share
            .language_statement()
            .encrypted_multiplicand()
            .value()
    })
    .collect();
```

- $\mathbf{ct}^i_2$ (**`encrypted_masked_key_shares`**): Resulting from the encryption of masked key shares using $\eta^i_{mask_1}$:

```
let encrypted_masked_key_shares: Vec<_> =
    ↪ masks_and_encrypted_masked_key_share
    .iter()
    .map(|mask_and_encrypted_masked_key_share| {
        mask_and_encrypted_masked_key_share
            .language_statement()
            .encrypted_product()
            .value()
    })
    .collect();
```

- **$\mathbf{ct}_3^i$ (`encrypted_nonces`)**: Produced by encrypting k_i using $\eta^i_{mask_3}$, essential for securely sharing nonce shares among participants:

```
let encrypted_nonces: Vec<_> =
    ↪ encrypted_nonce_shares_and_public_shares
    .iter()
    .map(|nonce_share_encryption_and_public_share| {
        nonce_share_encryption_and_public_share
            .language_statement()
            .encrypted_discrete_log()
            .value()
    })
    .collect();
```

The Encrypted Diffie-Hellman (EncDH) and Encrypted Discrete Logarithm (EncDL) public parameters facilitate the secure encryption and verification of these elements. Enhanced public parameters integrate range proofs.

The tuples $(\gamma_i, \eta^i_{mask_1}, \eta^i_{mask_2})$ and $(k_i, \eta^i_{mask_3})$ are mapped to their respective commitment structures, preparing them for secure multiparty computations. These mappings are used in commitment rounds to generate commitments and statements for zero-knowledge proofs, verifying operation correctness without revealing inputs:

```
let witnesses = mask_shares_witnesses
    .clone()
    .into_iter()
    .zip(
        masks_encryption_randomness
        .clone()
        .into_iter()
        .zip(masked_key_share_encryption_randomness),
    )
    .map(
        |(
            mask_share,
```

```
            (
                mask_share_encryption_randomness,
                masked_secret_key_share_encryption_randomness,
            ),
        )| {
            (
                mask_share,
                mask_share_encryption_randomness,
                masked_secret_key_share_encryption_randomness,
            )
            .into()
        },
    )
    .collect();
```

3.2.4 | Step 2b: Initialize Proof Aggregation

Relevant Code:

DIR src/presign/decentralized_party/encrypted_masked_nonces_round.rs

Step 2b of the protocol involves aggregating proofs related to the encrypted nonce shares and masked key shares, which is a continuation of the secure handling of cryptographic elements initiated in Step 2a.

The function `initialize_proof_aggregation` is responsible for setting up the proof aggregation necessary for the MPC aspects of the presign protocol.

The function begins by ensuring that the number of tuples $(\mathtt{ct}_1, \mathtt{ct}_2)$ provided matches the expected batch size:

```
if masks_and_encrypted_masked_key_share.len() != batch_size {
    return Err(Error::InvalidParameters);
}
```

This step extracts $\mathtt{ct}_1$, the encrypted masks, from each tuple:

```
let encrypted_masks: Vec<_> = masks_and_encrypted_masked_key_share
    .iter()
    .map(|statement| statement.encrypted_multiplicand().clone())
    .collect();
```

Here, the function samples randomness $(\eta^i_{mask_4})$, which is used later in the encryption of masked nonces, aligning with Step 2b (iii) of the protocol.

```
let masked_nonce_encryption_randomness =
    EncryptionKey::RandomnessSpaceGroupElement::sample_batch(
        self.encryption_scheme_public_parameters.
    ↪ randomness_space_public_parameters(),
        batch_size,
        rng,
    )?;
```

The function then maps tuples of encrypted masks, nonce shares, and their associated randomness into a new data structure for further processing. The following step involves generating cryptographic witnesses for each tuple. These witnesses are used to create commitments and statements essential for zero-knowledge proofs:

```
EnhancedLanguage::<
  SOUND_PROOFS_REPETITIONS,
  RANGE_CLAIMS_PER_SCALAR,
  COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS,
  RangeProof,
  UnboundedEncDHWitness,
  encryption_of_tuple::Language<
    PLAINTEXT_SPACE_SCALAR_LIMBS,
    SCALAR_LIMBS,
    GroupElement,
    EncryptionKey,
  >,
>::generate_witness(
  (
    nonce, // = k_i
    nonces_encryption_randomness, // = \eta^i_{mask_3}
    masked_nonces_encryption_randomness, // = \eta^i_{mask_4}
  )
    .into(),
  &enc_dh_public_parameters,
  rng,
)
.map_err(Error::from)
```

The final step involves setting up a commitment round where the encrypted data, along with their commitments, are processed to later generate $\texttt{ct}_4^i$ using `enhanced-maurer`:

```
enhanced_maurer::aggregation::commitment_round::Party::<
    SOUND_PROOFS_REPETITIONS,
    RANGE_CLAIMS_PER_SCALAR,
    COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS,
    RangeProof,
    UnboundedEncDHWitness,
```

```
    encryption_of_tuple::Language<
        PLAINTEXT_SPACE_SCALAR_LIMBS,
        SCALAR_LIMBS,
        GroupElement,
        EncryptionKey,
    >,
    ProtocolContext,
>::new_session(
    self.party_id,
    self.parties.clone(),
    enc_dh_public_parameters,
    self.protocol_context.clone(),
    vec![witness],
    rng,
)
.map_err(Error::from);
```

3.2.5 | Step 3: Centralized Party Verification of Presign Output

Relevant Code:

DIR src/presign/centralized_party/proof_verification_round.rs

This step involves verifying the outputs of the presigning process, specifically the encrypted components and their associated proofs. The function takes a batch of outputs, verifies each for consistency and correctness, and generates a verified output list.

3.2.5.1 | Input Validation

The function first ensures that the size of various batched outputs matches the expected batch size. This check ensures that the number of encrypted masks, encrypted masked key shares, and nonce public shares are all consistent, preventing any misalignment in the batch processing that follows.

```
if output.encrypted_masked_key_shares.len() != batch_size
    || output.encrypted_masks.len() != batch_size
    || output.nonce_public_shares.len() != batch_size
{
    return Err(Error::InvalidParameters);
}
```

3.2.5.2 | Decryption and Verification of Encrypted Data

The function processes encrypted data, including masks and masked key shares, by converting them into suitable cryptographic structures for verification. This is done using predefined public parameters from the encryption scheme. Each encrypted mask

is restructured using the ciphertext space public parameters to ensure it is correctly formatted for subsequent cryptographic operations.

```
let encrypted_masks = output
    .encrypted_masks
    .clone()
    .into_iter()
    .map(|encrypted_mask| EncryptionKey::CiphertextSpaceGroupElement::new
    ↪ (
        encrypted_mask,
        self.encryption_scheme_public_parameters.
    ↪ ciphertext_space_public_parameters(),
    ))
    .collect::<group::Result<Vec<_>>>()?;
```

3.2.5.3 | Verification of Range Proofs

Following decryption, the function verifies the range proofs associated with the key share masking process. The proofs for masks and encrypted masked key shares are verified against a set of statements that bind these elements together, ensuring that the encryption and the proof of correct encryption (range proof) are valid and consistent.

```
let statements = encrypted_masks
  .into_iter()
  .zip(encrypted_masked_key_shares)
  .zip(key_share_masking_range_proof_commitments)
  .map(
    |(
      (encrypted_mask, encrypted_masked_key_share),
      key_share_masking_range_proof_commitment,
    )| {
      (
        key_share_masking_range_proof_commitment,
        [encrypted_mask, encrypted_masked_key_share].into(),
      )
        .into()
    },
  )
  .collect();

output.masks_and_encrypted_masked_key_share_proof.verify(
    &self.protocol_context,
    &language_public_parameters,
    statements,
    rng,
)?;
```

3.2.5.4 | Handling of Nonce Public Shares

The function also processes nonce public shares, which are part of the output from the decentralized parties. Each nonce public share is processed to ensure it is in the correct cryptographic format, aligning with the group's public parameters.

```
let decentralized_party_nonce_public_shares = output
    .nonce_public_shares
    .clone()
    .into_iter()
    .map(|nonce_public_share| GroupElement::new(
        nonce_public_share, &self.group_public_parameters
    ))
    .collect::<group::Result<Vec<_>>>()?;
```

3.2.5.5 | Final Output Construction

Upon successful verification of all components and proofs, the function constructs the final verified presign output, encapsulating core elements like nonce shares, encrypted masks, and encrypted masked key shares. This segment constructs the final output for the presigning process, encapsulating all necessary cryptographic components required for the secure generation of a digital signature in a decentralized setting.

```
Ok(output
    .nonce_public_shares
    .into_iter()
    .zip(
        output.encrypted_masks.into_iter().zip(
            output.encrypted_masked_key_shares.into_iter().zip(
                self.signature_nonce_shares_and_commitment_randomnesses),
        ),
    )
    .map(
        |(
            decentralized_party_nonce_public_share,
            (
                encrypted_mask,
                (encrypted_masked_key_share, (nonce_share,
    ↪ commitment_randomness)),
            ),
        )| {
            Presign {
                nonce_share: nonce_share.value(),                    //
    ↪ = k_A
                decentralized_party_nonce_public_share,              //
    ↪ = R_B
                encrypted_mask,                                      //
    ↪ = ct_1
```

```
                encrypted_masked_key_share,                          //
    ↪ = ct_2
                commitment_randomness: commitment_randomness.value(), //
    ↪ = p_1
            }
        },
    )
    .collect())
```

3.3 | Sign Subprotocol

Protocol 6 (Figure A.3) in the 2PC-MPC paper [2] describes the sign subprotocol, which involves the generation of a decentralized digital signature. The protocol consists of several steps; the following sections summarize the functional correctness assessment of the sign subprotocol as realized in the provided Rust code.

3.3.1 | Step 1: Centralized Party Signature and Proof Setup

Relevant Code:

DIR `src/sign/centralized_party/signature_homomorphic_evaluation_round.rs`

> **!** **Out of Order Operations in Implementation**
>
> The implementation for Step 1, while functionally correct, strongly differs in ordering from the protocol steps as illustrated in the paper (Figure A.3)). This is due to the nature of the cryptographic operations and the need to optimize the code for performance and efficiency. In case the summary here is unclear, please refer to the actual code for a more detailed understanding.

The function `evaluate_encrypted_partial_signature_prehash` aims to evaluate the encrypted partial signature by utilizing homomorphic encryption and zero-knowledge proofs to ensure the security and integrity of the operations.

The nonce share (k_A) is inverted and multiplied by the decentralized party's public nonce share to compute the public nonce (R), as specified in Step 1a of the protocol:

```
let inverted_nonce_share = self.nonce_share.invert();
let public_nonce = inverted_nonce_share * self.
    ↪ decentralized_party_nonce_public_share;
```

Relevant values are also computed here, as part of a combined execution of Step 1b and 1e:

- U_A:

  ```
  let nonce_share_by_key_share_commitment = statement.
      ↪ altered_base_committment_of_discrete_log().clone();
  ```

- r:

  ```
  let nonce_x_coordinate = public_nonce.x();
  ```

- a_1:

```
let first_coefficient = (nonce_x_coordinate * self.nonce_share *
    ↪ self.secret_key_share) + (message * self.nonce_share);
```

- a_2:

```
let second_coefficient = nonce_x_coordinate * self.nonce_share;
```

Operations are performed relatively The L_{DComDL} proof for Step 1e dash 1 is then computed:

```
let (public_nonce_proof, _) = maurer::Proof::<
    SOUND_PROOFS_REPETITIONS,
    committment_of_discrete_log::Language<
        SCALAR_LIMBS,
        GroupElement::Scalar,
        GroupElement,
        Pedersen<1, SCALAR_LIMBS, GroupElement::Scalar, GroupElement>,
    >,
    ProtocolContext,
>::prove(
    &self.protocol_context,
    &language_public_parameters,
    vec![[self.nonce_share, self.nonce_share_commitment_randomness].into
    ↪ ()], // = [k_A, p_1]
    rng,
)?;
```

Then, a zero-knowledge proof of the public nonce's integrity is generated, using the previous commitment of discrete log parameters. This step secures the nonce's contribution to the signature, preventing adversaries from deriving the nonce or the private keys. This aligns with Steps 1d and 1e dash 3 of the protocol.

The following function performs homomorphic evaluations and generates proofs for these evaluations. These operations ensure that the encrypted signature component, s', adheres to the protocol's security requirements, providing a verifiable way to ensure the computations were performed correctly without revealing the underlying values.

```
let (encrypted_partial_signature_proof, statement) = enhanced_maurer::
    ↪ Proof::<
    SOUND_PROOFS_REPETITIONS,
    NUM_RANGE_CLAIMS,
```

```
    COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS,
    RangeProof,
    UnboundedDComEvalWitness,
    committed_linear_evaluation::Language<
        PLAINTEXT_SPACE_SCALAR_LIMBS,
        SCALAR_LIMBS,
        RANGE_CLAIMS_PER_SCALAR,
        RANGE_CLAIMS_PER_MASK,
        DIMENSION,
        GroupElement,
        EncryptionKey,
    >,
    ProtocolContext,
>::prove(
    &self.protocol_context,
    &language_public_parameters,
    vec![witness],
    rng,
)?;
```

Finally, the function packages the computed values and their corresponding proofs into a structure that will be used in subsequent verification steps of the protocol. This includes the public nonce, the encrypted partial signature, and all associated zero-knowledge proofs.

3.3.2 | Step 2: Decentralized Party Proof Verification and Signature Setup

In Step 2 of the protocol, the decentralized party verifies the encrypted partial signature and the associated zero-knowledge proofs. The party then proceeds to generate the encrypted partial signature and the corresponding zero-knowledge proof for the signature's homomorphic evaluation.

3.3.2.1 | Steps 2a, 2b: Signature Partial Decryption Round

Relevant Code:

DIR `src/sign/decentralized_party/signature_partial_decryption_round.rs`

The function `verify_encrypted_signature_parts_prehash_inner` is designed to verify the encrypted parts of a signature and associated zero-knowledge proofs in a secure multi-party computation setting.

The function begins by reconstructing the public nonce (R) from its components. This is followed by verifying the zero-knowledge proof of R, ensuring it was constructed correctly:

```
public_nonce_encrypted_partial_signature_and_proof
.public_nonce_proof
.verify(
    protocol_context,
    &language_public_parameters,
    vec![[
        centralized_party_nonce_share_commitment.clone(), // = K_A
        nonce_public_share.clone(),                       // = R_B
    ]
    .into()],
)?;
```

Similarly, the code verifies the ratio between committed values, ensuring that the relation $L_{\texttt{DComRatio}}$ described in [2] is maintained:

```
public_nonce_encrypted_partial_signature_and_proof
.nonce_share_by_key_share_proof
.verify(
    protocol_context,
    &language_public_parameters,
    vec![[
        centralized_party_nonce_share_commitment.clone(), // = K_A
        nonce_share_by_key_share_commitment.clone(),      // = U_A
    ]
    .into()],
)?;
```

Finally, the function generates a range proof commitment and bundles it into a call that also verifies the committed evaluations that compute the encrypted signature components, ensuring they meet the protocol's specified homomorphic properties, thereby implementing the last segment of Step 2a as well as Step 2b of the protocol simultaneously. The way this is done differs significantly from the paper: the zero-knowledge statements are reconstructed natively as described in Step 2b, and the statements sent by the other parties are not used:

```
let range_proof_commitment = proof::range::
    ↪ CommitmentSchemeCommitmentSpaceGroupElement::<
    COMMITMENT_SCHEME_MESSAGE_SPACE_SCALAR_LIMBS,
    NUM_RANGE_CLAIMS,
    RangeProof,
>::new(
    public_nonce_encrypted_partial_signature_and_proof
        .encrypted_partial_signature_range_proof_commitment,
    range_proof_public_parameters
        .commitment_scheme_public_parameters()
```

```
        .commitment_space_public_parameters(),
)?;

public_nonce_encrypted_partial_signature_and_proof
    .encrypted_partial_signature_proof
    .verify(
        protocol_context,
        &language_public_parameters,
        vec![(
            range_proof_commitment,
            (
                encrypted_partial_signature.clone(),
                [
                    ((nonce_x_coordinate *
    ↪ nonce_share_by_key_share_commitment)
                        + (message * &
    ↪ centralized_party_nonce_share_commitment)),
                    (nonce_x_coordinate * &
    ↪ centralized_party_nonce_share_commitment),
                ]
                .into(),
            )
                .into(),
        )
            .into()],
        rng,
    )?;
```

3.3.2.2 | Step 2c: Signature Threshold Decryption Round

Relevant Code[3]*:*

DIR `src/sign/decentralized_party/signature_threshold_decryption_round.rs`

The function `decrypt_signature` is tasked with the final decryption and verification of the signature components. This process is split into several key phases. The function starts by verifying that the set of decryption shares provided meets the expected criteria:

```
if decrypters.len() != usize::from(self.threshold) || ...
    return Err(Error::InvalidParameters);
```

The decryption shares are then combined to form the actual components of the signature:

[3]Supporting code may be found in `src/sign/decentralized_party/signature_partial_decryption_round.rs`.

```
// = pt_A
let partial_signature: Uint<PLAINTEXT_SPACE_SCALAR_LIMBS> =
    DecryptionKeyShare::combine_decryption_shares_semi_honest(
        partial_signature_decryption_shares,
        lagrange_coefficients.clone(),
        &self.decryption_key_share_public_parameters,
    )?
    .into();
let partial_signature = GroupElement::Scalar::new(
    partial_signature.reduce(&group_order).into(),
    &self.scalar_group_public_parameters,
)?;
```

The masked nonce, which is a product of the nonce and the key share, is derived and inverted:

```
 let masked_nonce: Uint<PLAINTEXT_SPACE_SCALAR_LIMBS> =
    DecryptionKeyShare::combine_decryption_shares_semi_honest(
        masked_nonce_decryption_shares,
        lagrange_coefficients,
        &self.decryption_key_share_public_parameters,
    )?
    .into();
let masked_nonce = GroupElement::Scalar::new(
    masked_nonce.reduce(&group_order).into(),
    &self.scalar_group_public_parameters,
)?;

// = (\gamma * k_B)^{-1}
let inverted_masked_nonce = masked_nonce.invert();
if inverted_masked_nonce.is_none().into() {
    return Err(Error::SignatureVerification);
}
```

This inversion is crucial for calculating the final signature scalar s. The signature scalar s is computed by multiplying the inverted masked nonce with the partial signature:

```
// s' = pt_4^-1 * pt_A
//    = k * (rx + m)
let signature_s = inverted_masked_nonce.unwrap() * partial_signature;
let negated_signature_s = signature_s.neg();
```

This computation directly follows the mathematical operations outlined in the protocol for final signature generation.

To prevent signature malleability, the function then computes s and $q - s$ and selects the smaller value:

```
let signature_s = if negated_signature_s.value() < signature_s.value() {
    negated_signature_s
} else {
    signature_s
};
```

This step ensures that the signature conforms to the standards required for ECDSA, where the scalar component of the signature must be within specific bounds.

Finally, the computed signature (r, s) is verified:

```
verify_signature(
    self.nonce_x_coordinate,
    signature_s,
    self.message,
    self.public_key,
)?;
```

4

Security Assessment

This chapter presents a security assessment of the provided Rust implementation of the 2PC-MPC protocol. The assessment focuses on identifying potential vulnerabilities, weaknesses, and deviations from best practices that could compromise the security of the cryptographic operations and the overall protocol.

The security assessment is based on a review of the codebase, with particular attention given to the handling of sensitive data, the implementation of cryptographic primitives, and the adherence to secure coding practices. The assessment aims to uncover any security issues that could lead to unauthorized access, data leakage, or the compromise of the protocol's integrity.

For each finding, the assessment provides detailed explanations of the issue, including relevant code snippets and the potential impact on the security of the protocol. Additionally, the assessment offers specific recommendations and mitigation strategies to address the identified vulnerabilities and strengthen the overall security posture of the 2PC-MPC implementation.

4.1 | **DW-01-001** Nonce Reuse in Decentralized Party Presigning Step

Severity: *Critical*
The reuse of nonces in the decentralized party presigning step has been identified as a critical security vulnerability. Nonce reuse in cryptographic operations can lead to a range of attacks, most notably allowing adversaries to potentially recover private keys or forge signatures.

The implementation of the presigning subprotocol samples a batch of mask shares for generating nonces (referred to as γ_i) and then reuses the same randomness for a different cryptographic purpose, namely generating signature nonce shares (referred to as k_i). The specific issue arises in the following code segment:

FILE `encrypted_masked_key_share_and_public_nonce_shares_round.rs`

```
let masks_shares = GroupElement::Scalar::sample_batch(
    &self.scalar_group_public_parameters,
    batch_size,
    rng,
)?;
let mask_shares_witnesses = masks_shares
    .clone()
    .into_iter()
    .map(|share| { EncryptionKey::PlaintextSpaceGroupElement::new(...).
    ↪ into() })
    .collect::<group::Result<Vec<_>>>()?;

// WARNING: This uses the same randomness as \gamma_i.
let shares_of_signature_nonce_shares_witnesses = masks_shares
    .clone()
    .into_iter()
    .map(|share| { EncryptionKey::PlaintextSpaceGroupElement::new(...).
    ↪ into() })
    .collect::<group::Result<Vec<_>>>()?;
```

This repeated use of the same randomness for both γ_i and k_i violates the fundamental cryptographic principle that nonces should be unique and unpredictable. Such practices can compromise the integrity and security of the cryptographic scheme.

Recommendation: Revise Nonce Handling Procedures

Modify the implementation to ensure that unique randomness is used for each cryptographic operation. This may involve generating new random values for each instance where a nonce is required. Additionally, conduct a thorough review of all cryptographic operations to ensure compliance with best practices concerning nonce management. Optionally, implement automated tests to check for unintended nonce reuse across the codebase.

4.2 | DW-01-002 Insufficient Checks on *ComputationalSecuritySizedNumber* Type

Severity: *Low*

The `ComputationalSecuritySizedNumber` type is implemented as a simple alias to `U128` without additional checks, potentially allowing values with insufficient entropy to be used in security-sensitive contexts.

Relevant Code:

DIR `group/src/lib.rs`

The `ComputationalSecuritySizedNumber` type is intended to represent values with sufficient computational security, typically requiring a minimum of 128 bits of entropy. However, the current implementation of `ComputationalSecuritySizedNumber` is a simple alias to the `U128` type, without any additional checks or restrictions on the values that can be assigned to it.

This lack of checks poses a potential security risk, as values with insufficient entropy could be inadvertently or maliciously cast to `ComputationalSecuritySizedNumber` and used in security-sensitive operations. For example, a `U64` value with only 64 bits of entropy could be cast to `ComputationalSecuritySizedNumber`, violating the intended security properties of the type.

Using values with insufficient entropy in cryptographic operations can weaken the overall security of the system. Attackers may be able to exploit this weakness to perform brute-force attacks, guess secret values, or compromise the integrity of the cryptographic primitives used in the 2PC-MPC protocol.

To address this issue, it is essential to enforce strict checks and validation on the values assigned to `ComputationalSecuritySizedNumber` to ensure that they meet the required entropy criteria. This can be achieved through a combination of compile-time and runtime checks.

Recommendation: Implement Strict Validation

Implement compile-time checks using Rust's type system to restrict the assignment of values to `ComputationalSecuritySizedNumber`. This can be achieved by creating a new type that wraps `U128` and provides a private constructor. The constructor should enforce the entropy requirements and only allow the creation of instances that meet the criteria. In addition, introduce runtime checks in the constructors and methods of `ComputationalSecuritySizedNumber` to validate the entropy of the input values. If a value fails to meet the entropy requirements, an error should be returned or the operation should be aborted. Finally, provide secure conversion methods between `ComputationalSecuritySizedNumber` and other primitive types, such as `U64`, that perform the necessary entropy checks. These conversion methods should be the only way to create instances of `ComputationalSecuritySizedNumber` from other types.

4.3 | DW-01-003 No Zeroization of Secrets in Memory

Severity: *Low*

The 2PC-MPC crate does not zeroize secrets in memory after they are no longer needed, potentially leaving sensitive data vulnerable to unauthorized access.

The 2PC-MPC crate performs various cryptographic operations that involve sensitive data, such as secret keys, nonces, and other private values. However, the crate does

not take any measures to securely erase these secrets from memory once they are no longer required. This lack of zeroization leaves the sensitive data vulnerable to potential unauthorized access or exploitation.

In the event of a memory disclosure vulnerability or a memory dump, an attacker could potentially access the unzeroized secrets, compromising the security of the cryptographic operations and the overall protocol. This is particularly concerning given the distributed nature of the 2PC-MPC protocol, where multiple parties are involved, and the security of the entire system depends on the confidentiality of the shared secrets.

The Rust implementation of 2PC-MPC involves numerous complex cryptographic operations and protocol steps, resulting in the generation and handling of a significant number of intermediate values. These intermediate values often contain sensitive information derived from the secret keys, nonces, and other private data used in the protocol.

Due to the high complexity of the codebase and the extensive use of generic types and traits, identifying and tracking all the intermediate values that require zeroization can be a challenging task. The intermediate values may be stored in various data structures, passed as function arguments, or returned as results, making it difficult to ensure comprehensive zeroization without a thorough analysis of the entire codebase.

To effectively implement zeroization in the 2PC-MPC crate, a comprehensive deep-dive into the codebase is necessary. This involves carefully examining each cryptographic operation and protocol step to identify all the intermediate values that contain sensitive information. Special attention should be given to the following aspects:

1. **Identifying Sensitive Data**: Analyze the codebase to identify all the data structures, variables, and function parameters that store or handle sensitive information, such as secret keys, nonces, and intermediate values derived from them.
2. **Tracing Data Flow**: Follow the flow of sensitive data throughout the codebase, including function calls, assignments, and data transformations, to ensure that all intermediate values are properly identified and tracked.
3. **Determining Intermediate Values Lifecycle**: Assess the lifecycle of each intermediate value to determine the appropriate point at which it should be zeroized. This includes identifying when the value is no longer needed and ensuring that zeroization occurs before the memory is deallocated or reused.
4. **Implementing Zeroization**: Modify the codebase to implement zeroization for all identified intermediate values. This may involve adding calls to the ‘zeroize()‘ method at the appropriate points, ensuring that the memory is securely erased before it is released or overwritten.

Conducting a comprehensive deep-dive into the 2PC-MPC codebase to identify and zeroize all intermediate values is a substantial undertaking. It requires a thorough understanding of the cryptographic operations, protocol steps, and the overall structure of the codebase.

Recommendation: Use the Rust `zeroize` crate

Utilize the Rust `zeroize` crate [8] to securely erase sensitive data from memory. The `zeroize` crate provides a simple and efficient way to zeroize memory locations that contain secrets. By implementing the `Zeroize` trait for sensitive data structures and calling the `zeroize()` method when the data is no longer needed, the crate ensures that the secrets are overwritten with zeroes, effectively preventing their recovery.

5

Conclusions

This report presented a comprehensive analysis and assessment of the provided Rust implementation of the 2PC-MPC protocol. Its focus has been on evaluating the functional correctness and security aspects of the implementation, with the goal of identifying potential issues, vulnerabilities, and areas for improvement.

5.1 | Summary of Core Assessments

The functional correctness assessment, presented in Chapter 3, involved a detailed examination of the three main subprotocols: the DKG subprotocol, the presign subprotocol, and the sign subprotocol. Implementations of each subprotocol were carefully reviewed, comparing them against the specifications outlined in the 2PC-MPC paper. The assessment found that the implementation generally aligns with the protocol descriptions, with the cryptographic operations and protocol steps being correctly realized in the Rust code. However, it is important to note that the assessment was conducted on a best-effort basis due to the high complexity of the target, and further analysis is recommended to ensure complete functional correctness.

The security assessment, covered in Chapter 4, focused on identifying potential vulnerabilities and weaknesses in the implementation that could compromise the security of the cryptographic operations and the overall protocol. Three main security findings of varying severity levels were reported. The critical issue of nonce reuse in the decentralized party presigning step highlights the importance of properly handling nonces to prevent attacks such as private key recovery or signature forgery. The medium severity issue of insufficient checks on the `ComputationalSecuritySizedNumber` type emphasizes the need for strict validation and entropy requirements for security-sensitive values. Lastly, the low severity issue of lack of zeroization of secrets in memory underscores the significance of securely erasing sensitive data to prevent unauthorized access.

5.2 | Note on Target Code Complexity

In addition to our core assessments, a notable observation during our analysis was the exceptionally high complexity of the Rust implementation. The 2PC-MPC protocol itself is intricate, involving multiple parties, cryptographic primitives, and protocol steps. The Rust implementation adds an additional layer of complexity due to its use of advanced language features, such as generics and traits, to encapsulate the cryptographic operations and protocol logic. While the use of these features provides flexibility and modularity, it can also make the codebase more challenging to understand, maintain, and audit. The complexity of the code increases the risk of introducing subtle bugs or vulnerabilities that may be difficult to detect and fix.

To address the complexity issue and improve the overall quality and security of the implementation, we recommend the following strategies for future development:

- **Code Simplification**: Wherever possible, aim to simplify the codebase by breaking down complex functions and structures into smaller, more manageable units. Use clear and descriptive naming conventions for variables, functions, and types to enhance code readability and understanding.
- **Comprehensive Documentation**: Provide detailed documentation for the codebase, including comments explaining the purpose and functionality of each component, as well as high-level descriptions of the protocol steps and cryptographic operations. Clear documentation will facilitate code review, maintenance, and future enhancements.
- **Rigorous Testing**: Implement a comprehensive testing strategy that covers both unit tests and integration tests. Write test cases that exercise all possible code paths and edge cases, ensuring the correctness and robustness of the implementation. Consider utilizing property-based testing techniques to automatically generate test cases and uncover hidden bugs.
- **Security Audits**: Conduct additional security audits on a regular schedule, focusing on the identification and mitigation of potential vulnerabilities. Engage external security experts to perform thorough code reviews and security assessments, providing an independent perspective on the implementation's security posture.
- **Community Engagement**: Foster an active and collaborative community around the 2PC-MPC implementation. Encourage open-source contributions, bug reports, and feedback from the wider cryptographic and security community. Engaging with the community will help identify and address potential issues, improve the implementation's quality, and promote its adoption and trust.

Our analysis of the provided Rust implementation of the 2PC-MPC protocol has highlighted both its strengths and areas for improvement. While the implementation

generally adheres to the protocol specifications, the high complexity of the codebase and the identified security findings warrant further attention and mitigation efforts. By addressing the complexity issue through code simplification, comprehensive documentation, rigorous testing, and regular security audits, the implementation can be enhanced to ensure its correctness, security, and maintainability.

We hope that the findings and recommendations presented in this report will serve as a valuable resource for the ongoing development and enhancement of the 2PC-MPC protocol and its Rust implementation. By addressing the identified issues and following best practices in secure software development, the implementation can be strengthened to provide a robust and trustworthy foundation for secure multi-party computation in various applications and domains.

5.3 | Future Work

While this report provides a comprehensive analysis of the functional correctness and security aspects of the provided Rust implementation of the 2PC-MPC protocol, there are several areas that warrant further exploration and development. In this section, we propose potential avenues for future work to enhance the protocol's security, efficiency, and usability.

5.3.1 | In-Depth Cryptographic Review

Although our assessment covered the functional correctness and security of the implementation, a more in-depth cryptographic review would be beneficial. This review should focus on the underlying cryptographic primitives, their theoretical foundations, and their suitability for the specific requirements of the 2PC-MPC protocol. A thorough analysis of the cryptographic constructions, including the choice of zero knowledge and Additively Homomorphic Encryption (AHE) primitives, encryption schemes, and zero-knowledge proof systems, would provide additional assurance regarding the protocol's security and help identify any potential weaknesses or improvements.

5.3.2 | Cryptographic Optimizations

The current implementation of the 2PC-MPC protocol relies on various cryptographic operations, such as homomorphic encryption, zero-knowledge proofs, and secure multi-party computation. While these cryptographic primitives provide strong security guarantees, they can also introduce computational overhead and impact the protocol's efficiency. Future work could explore optimizations to the underlying cryptographic constructions to improve performance without compromising security.

Potential optimizations include:

- Investigating alternative cryptographic primitives (zero knowledge constructions, AHE, etc.) representations and arithmetic that offer faster computation times while maintaining the required security level.
- Exploring more efficient homomorphic encryption schemes that reduce the computational cost of encryption and decryption operations.
- Optimizing the zero-knowledge proof systems to minimize the proof generation and verification times, as well as the size of the proofs.
- Implementing parallelization techniques to leverage multi-core processors and distribute the computational workload across multiple threads or machines.

By optimizing the cryptographic constructions, the 2PC-MPC protocol can achieve better performance, scalability, and practicality for real-world applications.

5.3.3 | Protocol API Correctness and Usability

The provided Rust implementation of the 2PC-MPC protocol can be viewed as a protocol in itself, with its own API and communication interfaces. Future work should focus on analyzing the correctness and usability of the protocol API to ensure that it is well-defined, consistent, and easy to use for developers and users.

This analysis should include:

- Reviewing the API documentation to ensure that it is clear, comprehensive, and accurately reflects the protocol's functionality and usage.
- Verifying the correctness of the API endpoints, input/output formats, and error handling mechanisms to prevent any inconsistencies or unexpected behavior.
- Assessing the ease of use and intuitiveness of the API, considering factors such as function naming conventions, parameter ordering, and default values.
- Conducting usability testing with developers and users to gather feedback on the API's design, documentation, and overall user experience.

By ensuring the correctness and usability of the protocol API, the 2PC-MPC implementation can be more easily integrated into various applications and systems, promoting its adoption and reducing the risk of implementation errors.

5.3.4 | State Machine Transition Analysis

The 2PC-MPC protocol involves multiple parties engaging in a sequence of steps and transitions to perform secure multi-party computation. To further enhance the protocol's security and reliability, future work should include a detailed analysis of the state machine transitions within the protocol.

This analysis should encompass:

- Formally defining the protocol's state machine, including all possible states, transitions, and conditions that trigger each transition.
- Verifying the correctness and completeness of the state machine, ensuring that it accurately represents the protocol's intended behavior and covers all possible scenarios.
- Analyzing the state machine for potential vulnerabilities, such as race conditions, deadlocks, or unhandled exceptions, that could compromise the protocol's security or disrupt its execution.
- Validating the implementation's adherence to the defined state machine, ensuring that the code correctly implements the specified transitions and handles all possible states and events.

By conducting a thorough state machine transition analysis, potential issues and vulnerabilities in the protocol's design and implementation can be identified and addressed, enhancing its overall security and reliability.

5.3.5 | Understanding the Decentralized Party in a Protocol Setting

In this audit report, we have primarily focused on the interaction between two main entities: the centralized party and the decentralized party. The centralized party is treated as a single, unified entity, while the decentralized party is considered as a single, virtually merged entity representing multiple participants. However, it is important to acknowledge that this simplification may not fully capture the nuances and complexities of the type of truly decentralized system that 2PC-MPC aims to facilitate.

In reality, the decentralized party consists of multiple individual parties, each with their own unique characteristics, behaviors, and potential for change. These parties may join or leave the system dynamically, merge with other parties, or undergo internal changes that affect their roles and responsibilities within the protocol.

To gain a more comprehensive understanding of the 2PC-MPC protocol and its implementation, future work should include an in-depth protocol analysis that treats each virtually decentralized party as a distinct individual party. This analysis should take into account the following considerations:

- **Party Composition**: Examine the composition of the decentralized party, considering the number of individual parties involved, their roles, and their relationships with each other. Analyze how the protocol handles the addition of new parties, the removal of existing parties, and the potential for parties to merge or split.
- **Communication Patterns**: Investigate the communication patterns among the individual parties within the decentralized party. Analyze how messages are exchanged, how consensus is reached, and how potential conflicts or disagreements are resolved. Consider the impact of network latency, asynchronous communication, and the possibility of malicious or faulty parties.
- **Key Management**: Explore the key management aspects of the protocol, focusing on how cryptographic keys are generated, distributed, and managed among the individual parties. Analyze the security implications of key sharing, key rotation, and the potential for key compromise or leakage.
- **Fault Tolerance**: Assess the protocol's resilience to failures or misbehavior of individual parties within the decentralized party. Examine how the protocol handles scenarios such as party disconnections, network partitions, or malicious actions. Evaluate the protocol's ability to maintain consistency, integrity, and availability in the presence of faults.
- **Scalability**: Investigate the scalability aspects of the protocol, considering how it performs as the number of individual parties within the decentralized party increases. Analyze the impact of larger party counts on communication overhead, computational complexity, and overall system performance.
- **Privacy and Security**: Assess the privacy and security guarantees provided by the protocol when treating each virtually decentralized party as an individual party. Analyze the potential for information leakage, collusion, or attacks that may arise from the interactions among individual parties. Evaluate the effectiveness of the protocol's privacy-preserving mechanisms and the robustness of its security properties.

5.3.6 | Performance and Scalability Testing

To assess the practical feasibility and scalability of the 2PC-MPC protocol, future work should include comprehensive performance and scalability testing. This testing should evaluate the protocol's behavior and performance under various conditions, such as different network latencies, varying numbers of participants, and increasing computational workloads.

The performance and scalability testing should cover:

- Measuring the protocol's throughput, latency, and resource utilization under different network conditions and participant configurations.

- Identifying performance bottlenecks and scalability limitations, such as network bandwidth constraints or computational resource exhaustion.
- Evaluating the protocol's ability to handle large-scale computations and data volumes, assessing its suitability for real-world applications.
- Comparing the performance and scalability of the 2PC-MPC protocol against other secure multi-party computation protocols or traditional centralized solutions.

By conducting thorough performance and scalability testing, the strengths and limitations of the 2PC-MPC protocol can be better understood, guiding future optimizations and improvements to enhance its practicality and adoption.

5.3.7 | Integration with Existing Systems and Frameworks

To facilitate the adoption and usage of the 2PC-MPC protocol, future work should explore its integration with existing systems and frameworks commonly used in secure computation and privacy-preserving applications.

This integration effort should include:

- Developing software development kits (SDKs) or libraries that provide high-level abstractions and simplified interfaces for integrating the 2PC-MPC protocol into existing applications and systems.
- Providing comprehensive documentation, tutorials, and examples to guide developers in integrating the 2PC-MPC protocol into their projects and utilizing its capabilities effectively.

By facilitating the integration of the 2PC-MPC protocol with existing systems and frameworks, its adoption and usage can be greatly enhanced, enabling developers and organizations to leverage secure multi-party computation in a wide range of applications and domains.

5.3.8 | Real-World Applications and Case Studies

To demonstrate the practical value and potential impact of the 2PC-MPC protocol, future work should focus on exploring real-world applications and conducting case studies in various domains.

Potential areas of application include:

- Privacy-preserving data analysis and machine learning, where sensitive data from multiple parties can be analyzed and models can be trained without revealing individual data points.

- Secure auctions and voting systems, where participants can engage in fair and transparent bidding or voting processes without disclosing their individual preferences.
- Collaborative financial computations, such as risk assessment or fraud detection, where financial institutions can jointly analyze data without sharing sensitive customer information.
- Secure supply chain management, where multiple parties can collaborate and optimize supply chain processes while preserving the confidentiality of their proprietary data.

By conducting case studies and demonstrating the successful application of the 2PC-MPC protocol in real-world scenarios, its practical feasibility, benefits, and potential for widespread adoption can be clearly showcased.

The proposed future work outlined in this section aims to further enhance the security, efficiency, usability, and practicality of the 2PC-MPC protocol and its Rust implementation. By addressing these areas, the protocol can be strengthened, optimized, and positioned as a valuable tool for secure multi-party computation in various domains, contributing to the advancement of privacy-preserving technologies and their real-world impact.

5.4 | Acknowledgments

Symbolic Software would like to thank Erik Takke and Tomer Ashur from 3MI Labs for their collaboration and support throughout the assessment process. We appreciate their expertise, insights, and dedication to advancing the field of secure MPC. We would also like to extend our sincere thanks to Yehonathan Cohen Scaly and Dolev Mutzari of dWallet Labs for their valuable contributions and feedback on the assessment findings. Their expertise and feedback have been instrumental in enhancing the quality and accuracy of the assessment results.

6

About Symbolic Software

Symbolic Software[1], established in Paris, France in 2017, is a software consultancy specializing in applied cryptography and software security. The firm has executed over 300 cryptographic software audits within the European information security sector and has made significant contributions to the field by publishing peer-reviewed cryptographic research software.

Offering wide-ranging expertise in cryptographic software audits, Symbolic Software has audited critical cryptographic components of global platforms, ranging from password managers to cryptocurrencies. The company has developed Verifpal® and Noise Explorer, innovative research software for cryptographic engineering, which have contributed to peer-reviewed scientific publications. Symbolic Software's portfolio is marked by collaboration with leading entities such as Cure53 and the Linux Foundation, and they have successfully audited critical technologies like MetaMask and key COVID-19 contact tracing applications in Europe.

[1] Stay updated on Symbolic Software's latest work by visiting `https://symbolic.sofware`.

Bibliography

[1] Ran Canetti et al. *UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts*. Cryptology ePrint Archive, Paper 2021/060. https://eprint.iacr.org/2021/060. 2021. DOI: 10.1145/3372297.3423367. URL: https://eprint.iacr.org/2021/060.

[2] Offir Friedman et al. *2PC-MPC: Emulating Two Party ECDSA in Large-Scale MPC*. Cryptology ePrint Archive, Paper 2024/253. https://eprint.iacr.org/2024/253. 2024. URL: https://eprint.iacr.org/2024/253.

[3] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[4] Ueli Maurer. "Unifying Zero-Knowledge Proofs of Knowledge." In: *Progress in Cryptology – AFRICACRYPT 2009*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 272–286. ISBN: 978-3-642-02384-2.

[5] Benedikt Bünz et al. "Bulletproofs: Short proofs for confidential transactions and more." In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[6] Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing." In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3.

[7] *Merlin Rust Crate Documentation*. https://docs.rs/merlin/latest/merlin/. Accessed: 2024-05-28.

[8] *Zeroize Rust Crate Documentation*. https://docs.rs/zeroize/latest/zeroize/. Accessed: 2024-05-29.

A

Main Subprotocol Figures

PROTOCOL 4 (*Key-Generation* Π_{keygen}*:* $\mathsf{KeyGen}(\mathbb{G}, G, q)$)

The protocol is parameterized with the ECDSA group description $(\mathbb{G}, G, q)$. The protocol interacts with $n+1$ parties $A, B_1, \ldots, B_n$, where all parties hold pk as input. The parties agree on a fresh sid and do as follows:

1. A**'s first message**:
   (a) A samples a random $x_A \leftarrow \mathbb{Z}_q$ and computes $X_A = x_A \cdot G$.
   (b) A sends $(\textbf{com–prove}, \mathsf{pid}_A, X_A; x_A)$ to $\mathcal{F}_{\mathsf{com-zk}}^{L_{\mathsf{DL}}}$.
2. B_i**'s first message**:
   (a) Each B_i receives $(\textbf{receipt}, \mathsf{pid}_A)$ from $\mathcal{F}_{\mathsf{com-zk}}^{L_{\mathsf{DL}}}$.
   (b) Each B_i samples a random $x_i \leftarrow \mathbb{Z}_q$ and computes $X_i = x_i \cdot G$.
   (c) *Only on first execution:* Each B_i sends $(\textbf{keygen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{TAHE}}$ and wait for the functionality's response. If the functionality responds with $(\textbf{pubkey}, \mathsf{sid}, \perp, U' \cap U)$ then output $U' \cap U$ and abort; otherwise, if the functionality responds with $(\textbf{pubkey}, \mathsf{sid}, pk)$ then continue.
   (d) Each B_i computes $\mathsf{ct}_i = \mathsf{AHE.Enc}(pk, x_i; \rho_i)$ for a randomly chosen ρ_i.
   (e) Each B_i sends $(\textbf{prove}, \mathsf{sid}, \mathsf{pid}_i, X_i, \mathsf{ct}_i; x_i, \rho_i)$ to $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$.
   (f) Each B_i receives $(\textbf{proof}, \mathsf{sid}, X_B, \mathsf{ct}_{\mathsf{key}})$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$, if not, it receives and outputs the set of corrupted parties and aborts.
3. A**'s second message**:
   (a) A receives $(\textbf{proof}, \mathsf{sid}, X_B, \mathsf{ct}_{\mathsf{key}})$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$. Note that A implicitly receives pk as well, as it is part of the public parameters of language L_{EncDL}.
   (b) A sends $(\textbf{certify}, pk)$ to $\mathcal{F}_{\mathsf{TAHE}}$ and waits for its response. If the response is 1 then continue, otherwise abort.
   (c) A sends $(\textbf{decom–proof}, \mathsf{pid}_A)$ to $\mathcal{F}_{\mathsf{com-zk}}^{L_{\mathsf{DL}}}$.
4. B_i**'s verification.**
   (a) Each B_i receives $(\textbf{decom–proof}, \mathsf{pid}_A, X_A)$ from $\mathcal{F}_{\mathsf{com-zk}}^{L_{\mathsf{DL}}}$, if not, it aborts.
5. **Output.**
   – A outputs $X = X_A + X_B$ and record $(\mathsf{keygen}, X_B, X, \mathsf{ct}_{\mathsf{key}}, pk)$.
   – Each B_i outputs $X = X_A + X_B$ and record $(\mathsf{keygen}, X_A, X, \mathsf{ct}_{\mathsf{key}}, pk)$.

Figure A.1: *"Protocol 4"* in the 2PC-MPC paper [2] covers the DKG key generation step.

PROTOCOL 5 (*Presigning* Π_{pres}*:* $\mathsf{Presign}((\mathbb{G}, G, q), \mathit{sid})$)

The protocol interacts with $A, B_1, \ldots, B_n$ where everyone has pk and $\mathsf{ct}_{\mathsf{key}}$ as input. Before proceeding, all parties verify that sid has never been used before. Then they do as follows:

1. A**'s message**:
   (a) A samples a random $k_A \leftarrow \mathbb{Z}_q$ and computes $K_A = \mathsf{Com}(k_A; \rho_1)$.
   (b) A sends $(\mathbf{prove}, \mathsf{sid}, \mathsf{pid}_A, K_A; k_A, \rho_1)$ to $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DCom}}}$.
2. B_i**'s message**:
   (a) *First round:*
      i. B_i receives $(\mathbf{proof}, \mathsf{sid}, \mathsf{pid}_A, K_A)$ from $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DCom}}}$, if not, it aborts.
      ii. B_i samples $k_i \leftarrow \mathbb{Z}_q^*$ and computes $R_i = k_i \cdot G$. Denote $k_B = \sum_i k_i$.
      iii. B_i samples $\gamma_i \in [0, q), \eta^i_{\mathsf{mask}_1}, \eta^i_{\mathsf{mask}_2}, \eta^i_{\mathsf{mask}_3}, \eta^i_{\mathsf{mask}_4} \in \mathcal{R}_{pk}$, and computes
         A. $\mathsf{ct}_1^i = \mathsf{AHE.Enc}(pk, \gamma_i; \eta^i_{\mathsf{mask}_1})$,
         B. $\mathsf{ct}_2^i = \mathsf{AHE.Eval}(pk, f_i, \mathsf{ct}_{\mathsf{key}}; \eta^i_{\mathsf{mask}_2})$ where $f_i(x) = \gamma_i \cdot x$,
         C. $\mathsf{ct}_3^i = \mathsf{AHE.Enc}(pk, k_i; \eta^i_{\mathsf{mask}_3})$.
      iv. B_i sends $(\mathbf{prove}, \mathsf{sid} \| \text{“}\gamma\text{”}, \mathsf{pid}_i, \mathsf{ct}_1^i, \mathsf{ct}_2^i; \gamma_i, \eta^i_{\mathsf{mask}_1}, \eta^i_{\mathsf{mask}_2})$ to $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_{\mathsf{key}}]}$.
      v. B_i sends $(\mathbf{prove}, \mathsf{sid}, \mathsf{pid}_i, R_i, \mathsf{ct}_3^i; k_i, \eta^i_{\mathsf{mask}_3})$ to $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$.
   (b) *Second round:*
      i. B_i receives $(\mathbf{proof}, \mathsf{sid} \| \text{“}\gamma\text{”}, \mathsf{ct}_1, \mathsf{ct}_2)$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_{\mathsf{key}}]}$ and $(\mathbf{proof}, \mathsf{sid}, R, \mathsf{ct}_3)$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$.
      ii. Otherwise, if B_i receives $(\mathbf{malicious}, \mathsf{sid}, U')$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_{\mathsf{key}}]}$ and/or $(\mathbf{malicious}, \mathsf{sid}, U'')$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$, it records the malicious parties (M and/or M') and aborts.
      iii. B_i computes $\mathsf{ct}_4^i = \mathsf{AHE.Eval}(pk, f_i', \mathsf{ct}_1; \eta^i_{\mathsf{mask}_4})$ where $f_i'(x) = k_i \cdot x$.
      iv. B_i sends $(\mathbf{prove}, \mathsf{sid} \| \text{“k”}, \mathsf{pid}_i, \mathsf{ct}_3^i, \mathsf{ct}_4^i; k_i, \eta^i_{\mathsf{mask}_3}, \eta^i_{\mathsf{mask}_4})$ to $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_1]}$.
   (c) *Proof verification:*
   B_i receives $(\mathbf{proof}, \mathsf{sid} \| \text{“k”}, \mathsf{ct}_3, \mathsf{ct}_4)$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_1]}$. Otherwise, if B_i receives $(\mathbf{malicious}, \mathsf{sid}, U')$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_1]}$, it records the malicious parties and aborts.
3. A**'s verification**
   (a) A receives $(\mathbf{proof}, \mathsf{sid}, R_B, \mathsf{ct}_3)$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDL}}}$, if not, it aborts.
   (b) A receives $(\mathbf{proof}, \mathsf{sid}, \mathsf{ct}_1, \mathsf{ct}_2)$ from $\mathcal{F}_{\mathsf{agg-zk}}^{L_{\mathsf{EncDH}}[pk, \mathsf{ct}_{\mathsf{key}}]}$, if not, it aborts.
4. **Output**
   (a) A records $(\mathsf{presign}, \mathsf{sid}, R_B, \mathsf{ct}_1, \mathsf{ct}_2; k_A, \rho_1)$, where ct_1 and ct_2 are encryptions of γ and $\gamma \cdot x_B$.
   (b) B_i records $(\mathsf{presign}, \mathsf{sid}, R_B, K_A, \mathsf{ct}_1, \mathsf{ct}_2, \mathsf{ct}_4)$, where ct_4 encrypts $\gamma \cdot k_B \mod q$.

Figure A.2: *"Protocol 5"* in the 2PC-MPC paper [2] covers the presigning step.

PROTOCOL 6 (*Signing* Π_{sign}*:* $\mathsf{Sign}((\mathbb{G}, G, q), \mathit{sid}, msg)$)

1. A**'s message**:
   (a) A computes $R = (k_A)^{-1} \cdot R_B (= k_A^{-1} k_B \cdot G)$ and $r = R|_{x-axis} \mod q$; denote $k = k_A^{-1} k_B$.
   (b) A samples $\rho_2 \in \mathcal{R}_{\mathsf{pp}}$ and computes $U_A = \mathsf{Com}(k_A \cdot x_A; \rho_2)$.
   (c) A sets $a_1 = r \cdot k_A \cdot x_A + m \cdot k_A$ and $a_2 = r \cdot k_A$.
   (d) A homomorphically evaluates $\mathsf{ct}_1, \mathsf{ct}_2$, on its private function $f_A(x_1, x_2) := a_1 x_1 + a_2 x_2$:
      – $\mathsf{ct}_A \leftarrow \mathsf{AHE.Eval}(pk, f_A, \mathsf{ct}_1, \mathsf{ct}_2; \eta_{\mathsf{eval}})$
   (e) A sends the following proofs:
      – A sends $(\mathbf{prove}, \mathsf{sid}, \mathsf{pid}_A, K_A, R_B; k_A, \rho_1)$ to $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComDL}}[\mathsf{pp}, (\mathbb{G}, R, q)]}$.
      – A sends $(\mathbf{prove}, \mathsf{sid}, \mathsf{pid}_A, K_A, U_A, X_A; k_A, x_A, \rho_1, \rho_2)$ to $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComRatio}}[\mathsf{pp}, (\mathbb{G}, G, q)]}$.
      – A computes $C_1 = (r \odot U_A) \oplus (m \odot K_A)$ and $C_2 = r \odot K_A$, and sends $(\mathbf{prove}, \mathsf{sid}, \mathsf{pid}_A, \mathsf{ct}_A, C_1, C_2; a_1, a_2, r \cdot \rho_2 + m \cdot \rho_1, r \cdot \rho_1, \eta)$ to $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComEval}}[\mathsf{pp}, pk, \mathsf{ct}_1, \mathsf{ct}_2]}$.
2. B_i**'s verification and output**:
   (a) B_i receives the following proofs, otherwise, it aborts.
      – $(\mathbf{proof}, \mathsf{sid} || \mathsf{pid}_A, K_A, R_B)$ from $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComDL}}[\mathsf{pp}, (\mathbb{G}, R, q)]}$.
      – $(\mathbf{proof}, \mathsf{sid} || \mathsf{pid}_A, K_A, U_A, X_A)$ from $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComRatio}}[\mathsf{pp}, (\mathbb{G}, G, q)]}$.
      – $(\mathbf{proof}, \mathsf{sid} || \mathsf{pid}_A, \mathsf{ct}_A, C_1, C_2)$ from $\mathcal{F}_{\mathsf{zk}}^{L_{\mathsf{DComEval}}[\mathsf{pp}, pk, \mathsf{ct}_1, \mathsf{ct}_2]}$.
   (b) B_i verifies that the values used in the proofs are consistent with values obtained previously by B_i, specifically:
      – There are records $(\mathsf{keygen}, X_A, X, \mathsf{ct}_{\mathsf{key}}, pk)$ and $(\mathsf{presign}, \mathsf{sid}, R_B, K_A, \mathsf{pt}')$, and
      – $C_1 = (r \odot U_A) \oplus (m \odot K_A)$ and $C_2 = r \odot K_A$, where $r = R|_{x-axis}$.
   (c) B_i sends $(\mathbf{decrypt}, pk, \mathsf{ct}_A)$ and $(\mathbf{decrypt}, pk, \mathsf{ct}_4)$ to $\mathcal{F}_{\mathsf{TAHE}}$ and waits for its response. Let the responses be $(\mathbf{decrypted}, pk, \mathsf{ct}_A, \mathsf{pt}_A, U_A)$ and $(\mathbf{decrypted}, pk, \mathsf{ct}_4, \mathsf{pt}_4, U_4)$ respectively; if $\mathsf{pt}_A = \perp$ or $\mathsf{pt}_4 = \perp$ then output $U_A \cup U_4$ and abort; otherwise compute $s' = \mathsf{pt}_4^{-1} \cdot \mathsf{pt}_A \mod q$ (which is equal to $(\gamma k_B)^{-1} \cdot ((r k_A x_A + m k_A)\gamma + r k_A \gamma x_B) = k^{-1}(rx + m) \mod q$ as required) and output $s = \min\{s', q - s'\}$ (to ensure uniqueness of the signature).
3. **Output:** B_i outputs $\sigma = (r, s)$.

Figure A.3: *"Protocol 6"* in the 2PC-MPC paper [2] covers the generation of the actual decentralized signature.