Garbling-based MPC Protocol Security Analysis Report

Customer: COTI and Soda Labs

Date: 04/08/2025

We express our gratitude to the COTI and Soda Labs team for the collaborative engagement that enabled the execution of this Garbling-Based MPC Protocol Security Assessment.

The MPC protocol leverages garblers and dual evaluators to privately compute smart contracts, preserving privacy and security for Ethereum Virtual Machine (EVM) execution in a blockchain setup.

Document

Name	Garbling-Based MPC Protocol Review and Security Analysis Report for COTI and Soda Labs
Audited By	Mike Mu
Approved By	Nino Lipartiia
Language	C++
Tags	Garbled Circuit, MPC, Secure Computation
Methodology	https://hackenio.cc/blockchain_methodology

Review Scope

Repository	https://github.com/soda-mpc/soda-mpc
Commit	7fb3a27404df5720bf557eca2f0e19cd56081458
Commit After Fixes	649eb99e44061b06cad77acbe626f6edded53065

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

55	53	2	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	3
High	20
Medium	15
Low	8

Vulnerability	Severity
F-2025-8671 - Insufficient File Path Validation in writeToFile and readVectorFromFile Functions, Exposing to Directory Traversal Attacks	Critical
F-2025-8886 - Failure to Enforce Minimum Entropy Requirements	Critical
F-2025-9152 - Weak Randomness, Potential Buffer Overflow in Commons::generateDeviceRandomBytes	Critical
F-2025-8667 - Buffer Overflow Risk in Commons::convertHexToBytes	High
F-2025-8691 - Possible Memory Leaks With Context	High
F-2025-8692 - Potential Race Condition in waitForClient() Function	High
F-2025-8815 - Side-Channel Vulnerability Due to Signal Bit Manipulation	High
F-2025-8818 - Risk of Integer Overflow Across Multiple Files	High
F-2025-8827 - Non-Constant Time Operations in Signal Bit Comparison	High
F-2025-8884 - Absence of Message Size Validation in RSA Encryption	High
F-2025-8938 - Insufficient Input Validation	High
F-2025-8939 - Absence of Secure Memory Wiping in Destructor	High
F-2025-8941 - Potential Underflow in OpenSSLHash::hash()	High
F-2025-8946 - Absence of Null Check Following Memory Allocation	High
F-2025-8947 - Memory Management Issues in Garbler::generateBatch()	High
F-2025-8948 - Race Condition Due to Unsynchronized Access to exitThreads	High
F-2025-8951 - Non Atomic And Non Join Safety In Destructor	High
F-2025-8968 - Unsafe Exception Handling in GCManager::refillCapacities	High
F-2025-8970 - Unsafe Destructor Implementation in GCManager	High

Vulnerability	Severity
F-2025-9031 - Potential Memory Leak in GarblerServer	High
F-2025-9173 - Resource Leak in setServerAddress Function	High
F-2025-9174 - Incomplete Cleanup in Destructor	High
F-2025-9195 - Exposure of Internal State through Raw Pointers	High
F-2025-8669 - Resource Cleanup Failure on Exception During Allocation	Medium
F-2025-8698 - Memory and Exception Handling Issues in BooleanCircuit Constructor	Medium
F-2025-8807 - Memory Leak in garble() Function Due to Unfreed Allocations	Medium
F-2025-8811 - Memory Leak if indexArray Allocation Fails	Medium
F-2025-8813 - Memory management in garbleOutputWiresToNoFixedDelta	Medium
F-2025-8814 - Potential Integer Overflow in getGarbledTableSize	Medium
F-2025-8817 - Absence of Input Validation	Medium
F-2025-8825 - Potential Memory Leak in Constructor of HalfGatesCircuitEvaluator	Medium
F-2025-8828 - Unsafe Array Access Leading to Potential Crash	Medium
F-2025-8829 - Potential Memory Leak in Constructor of HalfGatesCircuitGarbler	Medium
F-2025-8831 - Questionable Memory Cleanup on halfGatesCircuitGarbler.cpp	Medium
F-2025-8940 - Sensitive Memory Data Not Properly Cleared in Destructor	Medium
F-2025-9033 - Missing Channel Closure Mechanism in GarblerClient and GCClient	Medium
F-2025-9197 - Hardcoded FixedKey Violates Best Practices	Medium
F-2025-9219 - Memory Leak in Constructor due to Unreleased Resources	Medium
F-2025-8689 - Potential Race Condition Between Directory Existence Check and Creation	Low
F-2025-8694 - Unsafe File Operation in readCircuitFromFile	Low
F-2025-8808 - Side-Channel Information Leak	Low
F-2025-8809 - Side-Channel Information Leak in Signal Bit Computation	Low
F-2025-8812 - Potential Integer Overflow in getGarbledTableSize	Low
F-2025-8824 - Lack of Error Handling in AES Encryption Functions	Low
F-2025-8883 - Missing Input Validation in OpenSSL_ECDSA::sign()	Low
F-2025-8928 - Absence of Secure Memory Data Wiping	Low
F-2025-8826 - Missing Input Validation Leading to Potential Buffer Overflow	Info
F-2025-8838 - Missing Documentation on Security Assumptions in TedKrovetzAesNiWrapperC.cpp	Info
F-2025-8887 - DER Format Validation Issues	Info
F-2025-8937 - Unsafe Casting Using const_cast on String Literals	Info
F-2025-8986 - Insecure Channel	Info
F-2025-9178 - Non-Idiomatic Exception Handling in gRPC Service Method	Info
F-2025-9193 - Unimplemented getInputWireIndices Function	Info
F-2025-9216 - Incomplete Verification in internalVerify	Info
F-2025-9217 - Insufficient Documentation	Info

Documentation quality

- Documentation is accessible within the codebase, offering clear explanations for components like input wire types and method functionalities.
- Efforts to articulate intent are evident, providing a solid starting point for a complex cryptographic system.
- The client provides a comprehensive whitepaper, "Soda Labs MPC Specification for: A Protocol for Two Evaluators" (December 2024), detailing the system design, including security assumptions for IT-MACs, label authentication, and high-level protocol overviews for both semi-honest and malicious threat models. This enhances the effectiveness of security analysis, though minor gaps in implementation specifics remain.
- README and building instructions are present but outdated, requiring revision to reflect current developments.
- Recommendations include completing TODOs, adding security notes, unifying comment depth, and providing a protocol summary.

Code quality

- Codebase demonstrates high potential with advanced techniques like Half Gates and Free XOR for private smart contract computation.
- Core cryptographic logic shows thoughtful design.
- Issues include unowned pointers, hardcoded values, and missing input validation/buffer checks.
- Non-constant-time operations and unresolved TODOs highlight areas needing attention for robustness.
- Modularity is maintained, but edge cases and integrity checks require further specification to ensure security.

Architecture quality

- Architecture leverages innovative separation and distribution of the garbling and evaluation design with Half Gates and Free XOR, enhancing privacy and decentralization.
- Cryptographic whitepaper provides a clear foundation, supporting maintainability of the garbling process.
- The Garbler’s central role in generating garbled circuits and reliance on fixed cryptographic seeds for key derivation suggest opportunities for dynamic key management enhancements, such as periodic key rotation or distributed key generation, to enhance security and resilience.

Table of Contents

System Overview

Soda MPC Protocol is designed to facilitate privacy-preserving execution of user-defined smart contracts. Built on Yao's Garbled Circuits with Half Gates optimization and Free XOR techniques, it operates as an MPC-as-a-service platform. Users onboard with a symmetric encryption key, shared among MPC nodes, to encrypt private transaction data, ensuring confidentiality even from individual nodes. The protocol supports malicious mode - tailored to different threat models, aiming to guarantee correctness and privacy under limited adversarial corruption.

System Architecture

The architecture of the version under this audit revolves around a distributed setup involving two Garblers with different signing keys, two Evaluators, and a Key Management Service (KMS). These components collaborate across offline and online phases to prepare, distribute, and compute garbled circuits representing EVM opcodes of encrypted arguments. The design emphasizes privacy by splitting sensitive data and computation across parties, with no single node gaining full visibility into user inputs or outputs.

Garbler Replicas

Role: Acts as the circuit generators and coordinators, responsible for transforming EVM opcodes into a garbled form that obscures its execution.

Function: A Garbler produces garbled circuits using a randomized process, incorporating optimizations like Half Gates to reduce computational overhead. It generates and distributes shares of input labels, permutation bits, and authentication data to the Evaluators. To protect against a single malicious Garbler, two Garblers operate as replicas to protect against single-point compromise, ensuring consistency in circuit generation.

Evaluators (E_1 and E_2)

Role: Serve as the computational engines, jointly evaluating garbled circuits to produce contract outputs without learning the underlying private data.

Function: Receive garbled circuits and shares from the Garbler, reconstruct necessary labels based on input types (public, private prior outputs, or secret-shared), and verify the authenticity of data using cryptographic checks. The dual-evaluator setup ensures that each party holds only partial information, requiring cooperation to complete the computation and maintain privacy.

Key Management Service (KMS)

Role: Initiating user key generation and authentication, mitigating risks from dishonest Evaluators.

Function: Generates pseudorandom key shares for users, encrypts them for delivery, and provides authentication data to Evaluators to ensure key consistency across uses. Typically co-located with the Garbler, it acts as a trusted intermediary to prevent Evaluators from supplying incorrect shares or disrupting the protocol. In order to mitigate a single malicious KMS node, this function is replicated twice and the Evaluators match their response before proceeding to the next steps.

Major Components and Workflow

- **Garbled Circuits:** Represent EVM opcodes as cryptographic constructs, using Half Gates to optimize AND gates (two ciphertexts per gate) and Free XOR to eliminate XOR gate overhead. The Garbler creates these circuits offline, embedding a per circuit offset (Δ) to maintain label relationships.
- **Label and Bit Sharing:** Inputs and outputs are represented by labels (e.g., L_{i,0}, L_{i,1}) and permutation bits (p_i), split into shares between Evaluators. This ensures no single party can encode or decode private values without collaboration.
- **Authentication Mechanisms:** Information-theoretic MACs (IT-MACs) secure permutation bits, leveraging XOR-homomorphism for efficient verification, while AES-based PRP authenticate labels, ensuring integrity against tampering.
- **User Key Management:** Users receive a symmetric AES key (k_U) during onboarding, split into shares held by Evaluators. In the semi-honest variant, Evaluators generate and deliver shares directly; in the malicious variant, the KMS ensures authenticated, consistent shares.

Protocol Phases

- **Offline Phase:** The Garbler prepares garbled circuits and distributes them, along with shares and authentication data, to the Evaluators. This phase uses pseudorandom generation from seeds to enable matching of GCs from Garbler's replicas, enabling scalable circuit preparation.
- **Online Phase:** Evaluators consume garbled circuits as needed, reconstructing labels for current inputs, verifying authenticity, and evaluating the circuit to compute outputs. The process supports public, private, and secret-shared inputs, adapting to each type’s privacy requirements.
- **User Onboarding:** Users join the system by obtaining a symmetric key, shared among Evaluators, which they use to encrypt transaction data.

Threat Models and Security Goals

- **Malicious Variant (2 Garblers, 2 Evaluators):** Handles active corruption of one Garbler or Evaluator, maintaining correctness and privacy (though availability may suffer); multiple corruptions can compromise both properties.

Summary

The Garbling-based MPC Protocol enables private smart contract execution by distributing trust across Garblers and Evaluators, with optional KMS support for malicious settings. It leverages optimized garbling, shared labels, and robust authentication to balance privacy, efficiency, and security, making it a specialized MPC solution for decentralized applications.

Risks

Below is a summarized list of risks associated with the Garbling-based MPC Protocol with two Garbler replicas and two Evaluators, considering the scenario that stands in contrast to Soda Labs' recommendation, where a single entity operates the Garbler, Evaluator 1 (E_1), and Evaluator 2 (E_2) in good faith. This assessment integrates insights from the Soda Labs whitepaper with protocol specifications therein (December 2024), the code implementation, and the operational context. Each risk is evaluated based on its potential impact on security, privacy, and correctness, assuming the entity's good-faith intent but acknowledging design, implementation, and operational factors.

Single Point of Failure in Operational Resilience

Description: With one entity running all nodes (Garbler, E_1, E_2, and KMS), which is a operational mode not recommended by Soda Labs, a system failure, outage, or compromise affects the entire protocol. The whitepaper notes availability risks in the malicious variant if one node stalls (Section 1), amplified here by centralized operation.

Evaluation: The code lacks fault-tolerance mechanisms (e.g., no retry logic in ProtocolPreDeterminedRounds), and the whitepaper's deployment details (Section 7) are deferred, suggesting unaddressed resilience planning. A hardware failure, DDoS attack, or software bug could halt service, as there's no redundancy across entities. Good faith doesn't prevent operational risks.

Key Management

Description: The protocol employs a Key Management Service (KMS) to facilitate authenticated user key generation (Section 5.2), ensuring secure onboarding of users while maintaining the distributed nature of key shares across Evaluators.

Evaluation: The KMS generates a temporary user key share (k_U') and IT-MACs (e.g., M_1[k_U,1,j'], M_2[k_U,2,j']) using its secrets S, GMK_1, and GMK_2, enabling Evaluators E_1 and E_2 to authenticate key shares during user onboarding (Section 5.2, steps 3-4). Evaluators independently compute k_U*, ensuring the final user key (k_U = k_U' ⊕ k_U*) remains distributed and known only to the user, aligning with the protocol's goal of preventing any single node from accessing the full key (Section 5). To enhance security, KMS replicas share the same secrets but are monitored by Evaluators, who verify the integrity of KMS responses (e.g., signed messages, step 4) and halt the protocol if inconsistencies are detected (e.g., mismatched receipts, step 5), triggering a manual check to address potential compromises. While this mechanism helps prevent immediate misuse of forged IT-MACs in the current session, a compromise of a KMS replica may still expose S, GMK_1, and GMK_2, increasing the risk of IT-MACs being forged in other contexts.

Potential for Insider Threats, Misconfiguration and Single-Entity Control

Ideal Deployment Scenario (Distributed Entities)

Description: In the ideal deployment, the Garbler, Evaluators (E_1, E_2), and KMS are operated by distinct entities, aligning with the whitepaper's distributed trust model (Section 1). Security claims ensure correctness and privacy with at most one node corrupted, limiting the impact of insider threats or misconfigurations.

Evaluation: The protocol splits secrets across entities (e.g., k_U = k_{U,1} ⊕ k_{U,2}, with k_{U,1} at E_1 and k_{U,2} at E_2, Section 5; labels L_i^b = R_{i,1}^b ⊕ R_{i,2}^b, Section 3.1), ensuring no single entity can reconstruct private data. An insider at E_1 can access k_{U,1}, P_2[k_{U,1,j}], and GMK_1 (Section 5.2), but cannot forge IT-MACs without GMK_2 (held by E_2) or decrypt user data without k_{U,2}. A misconfiguration, such as over-permissive access to ek_1 (AES key shared with E_1, Section 6.3), might expose encrypted shares, but the distributed model limits the impact. However, the whitepaper lacks audit trails or role-based access controls (RBAC) to mitigate insider risks within each entity. For example, a rogue employee at the Garbler could leak S_G (Section 6.2), potentially aiding side-channel attacks, though not directly breaking privacy. External coercion (e.g., legal pressure on one entity) or collusion between entities (e.g., Garbler and E_1) could also combine shares (S_1, k_{U,1}), simulating multi-node corruption, but this is outside the protocol's threat model.

Worst-Case Scenario (Single Entity)

Description: When a single entity operates all nodes (Garbler, Evaluators, KMS), it can access all shares (e.g., R_{i,1}^b, R_{i,2}^b, p_{i,1}, p_{i,2}, Section 3.1) and keys (GMK_1, GMK_2, k_{U,1}, k_{U,2}, Section 5.2), negating the whitepaper's distributed trust assumptions and exposing the system to insider threats, misconfigurations, or architectural compromise.

Evaluation: A single entity can reconstruct all secrets, such as k_U = k_{U,1} ⊕ k_{U,2}, L_i^b = R_{i,1}^b ⊕ R_{i,2}^b, and p_i = p_{i,1} ⊕ p_{i,2}, effectively simulating the worst-case corruption scenario in the malicious variant (e.g., both Garblers or Evaluators corrupted, Section 1), which breaks correctness and privacy. An insider (e.g., a disgruntled employee) could access these secrets to decrypt user data or forge IT-MACs (e.g., M_1[k_{U,1,j}]), bypassing authentication. A misconfiguration, such as exposing the KMS's S due to poor access controls, would allow regeneration of all user key shares (k_U') and, combined with Evaluator shares (k_U*), fully compromise user keys. Even in good faith, an internal breach or external coercion (e.g., legal mandate) could expose user data, as no cryptographic separation enforces trust distribution. The lack of audit trails or RBAC exacerbates this risk, as there's no way to detect or limit insider actions, making a single breach catastrophic.

Scalability and Performance Considerations

Description: The protocol's current design does not explicitly implement sharding or parallelization (whitepaper Section 6.1 focuses on bandwidth optimization). However, the protocol does not preclude parallelization, suggesting potential for future scalability improvements.

Evaluation: The workload of garbling and evaluation is distributed across the Garbler and Evaluators, which helps balance resource demands as user demand grows, though the lack of explicit parallelization mechanisms within each role may still limit computational throughput for large-scale operations. The protocol leverages the Half Gates scheme (Appendix A) for gate optimization reportedly offering the best performance for privacy-preserving EVMcompatible MPC to date, as per the client’s performance documentation from the go-ethereum audit. Specific implementation details, such as unoptimized loops in MAC generation (e.g., IT-MAC computations in Section 6.3) and memory copies (e.g., in garbled wire assignments), may still degrade performance, particularly for large circuits. Addressing these bottlenecks and incorporating parallelization could further enhance scalability and efficiency for broader adoption.

COTI's Team Response

> *According to the COTI team, they have introduced changes to address operational and architectural risks identified in the initial audit, particularly those related to trust concentration and single-entity control. They state that node management has been transitioned to a distributed model to reduce single points of failure and to incorporate redundancy as recommended by Soda Labs. The team mentioned that, during earlier pre-mainnet deployments, components such as the Garbler, Evaluators, and KMS were operated by a single entity for development and testing purposes. As stated by the team this is no longer the case in the current production environment. They also indicate plans to continue decentralizing operations by separating responsibilities among independent parties, expanding redundancy, and improving monitoring.*

Findings

Vulnerability Details

F-2025-8671 - Insufficient File Path Validation in writeToFile and readVectorFromFile Functions, Exposing to Directory Traversal Attacks - Critical

Description:

File locations:

- *lib/src/commons/commons.cpp*
- *lib/src/communication/communication.cpp*
- *lib/src/crypto/circuit/booleanCircuit.cpp*
- *lib/src/crypto/circuit/garblingCircuit.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*
- *lib/src/crypto/digitalSignature/digitalSignature.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolEvaluator.cpp*
- *opcodesTranslator/opcodesToMPC.cpp*
- *opcodesTranslator/userCommandsHandler.cpp*

Incomplete Validation:

The function only checks for `..` in the file path, a common indicator of directory traversal, but this is insufficient to fully prevent such attacks.

Attackers may bypass this check using techniques such as:

- Using absolute paths (e.g., `/etc/passwd` on Unix-like systems or `C:\Windows\System32\` on Windows).
- Using URL-encoded or Unicode-encoded characters (e.g., `%2e%2e` for `..`).

Lack of Path Sanitization:

The function does not normalize or sanitize the file path, which is essential to ensure the path is valid and safe.

```
bool Commons::isValidFilePath(const string& filepath) {
// Check for directory traversal attempts
if (filepath.find("..") != string::npos) {
return false;
}
// Add additional path validation as needed
return true;
}
void Commons::writeToFile(string fileName, string content, bool app
end)
{
if (!isValidFilePath(fileName)) {
Logger::getInstance().critical("Invalid file path: " + fileName);
throw runtime_error("Invalid file path");
}
// ... rest of existing code ...
}
```

File location: *lib/src/commons/logger.cpp*

```
void Logger::setName(string name, string pathToConfigFile) {
// ... existing code ...
auto config = Commons::readTomlFile(pathToConfigFile + "config.toml
"); // Vulnerable to path traversal
// ... existing code ...
string trace_name = "logs/" + name + "_logTrace_" + stringTime.str(
) + ".txt"; // Vulnerable to path traversal
string warn_name = "logs/" + name + "_logCritical_" + stringTime.st
r() + ".txt"; // Vulnerable to path traversal
}
```

The code concatenates paths and filenames without sanitization. An attacker could potentially use special characters in the name or `pathToConfigFile` to access unauthorized files.

This is a critical-severity item, as it can lead to a complete breakdown of the blockchain network's security and privacy by enabling unauthorized access to sensitive files.

Impact: Incomplete validation and lack of path sanitization in `Commons::isValidFilePath` allow directory traversal attacks, letting attackers read/write sensitive files (e.g., */etc/passwd*, garbler keys), compromising node security and protocol integrity.
Likelihood: High—file inputs (e.g., *pathToConfigFile*, name) are attacker-controllable in multi-party or misconfigured setups; simple bypasses (e.g., */etc/passwd, %2e%2e*) exploit the weak check.

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Severity: Critical

Recommendations

Remediation: Consider the following measures to enhance security and reliability when handling file paths:

- **Normalize the Path** – Convert the path to a standard format to eliminate inconsistencies.

- **Restrict to a Base Directory** – Ensure that file operations are confined within a predefined directory.
- **Validate Path Components** – Sanitize individual path components to prevent directory traversal attacks.

Resolution: The issue is fixed with these 2 commits:

https://github.com/soda-mpc/soda-mpc/commit/b39e3d92f688e408c3d83165608acf1d96611c0d

https://github.com/soda-mpc/soda-mpc/commit/39d37c65e2eef6915aaf0b39ba386e8ef4efaf00

F-2025-8886 - Failure to Enforce Minimum Entropy Requirements - Critical

Description: File: *lib/src/crypto/encryption/cryptoppRsa.cpp*

The code only issues a warning for insufficient entropy and proceeds with a weak random seed, which may compromise security. HMAC requires sufficient entropy to maintain its security.

This is a critical-severity item, as it can lead to a complete breakdown of the blockchain network's security and privacy by undermining cryptographic integrity.

Impact: Using a weak random seed (<32 bytes) in `CryptoPP_RSAOAEP::encrypt` compromises HMAC entropy, generating predictable ciphertexts or keys, allowing attackers to decrypt messages (e.g., garbled tables) or forge signatures, breaking privacy and correctness.
Likelihood: High—insufficient entropy is plausible (e.g., misconfigured RNG, low-entropy source); proceeding without enforcement ensures vulnerability.

```
void CryptoPP_RSAOAEP::encrypt(...) {
if (randomSize < 32){
Logger::getInstance().warn("Random seed size is less than 32 bytes"
);
}
// Continues with weak seed
}
```

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Severity: Critical

Recommendations

Remediation: Interrupt the program if the entropy falls below the minimum required threshold.

```
void CryptoPP_RSAOAEP::encrypt(...) {
const size_t MIN_ENTROPY = 32; // 256 bits minimum entropy
if (randomSize < MIN_ENTROPY) {
Logger::getInstance().critical("Insufficient random seed size");
```

```
  throw invalid_argument("Random seed must be at least 32 bytes");
  }
  ...
  }
```

Resolution:

The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/c885ed5871f6bb5253b936b6e09df8558af4b828

F-2025-9152 - Weak Randomness, Potential Buffer Overflow in Commons::generateDeviceRandomBytes - Critical

Description:

File: *lib/src/commons/commons.cpp:258*

Several critical and high-severity issues have been identified in the `Commons::generateDeviceRandomBytes()` function:

Issue 1: Use of Non-Cryptographicly Secure Randomness

The function relies on `std::random_device`, which is not guaranteed to be cryptographically secure across all platforms. On certain platforms (e.g., MinGW), it can be deterministic, making it unsuitable for cryptographic applications.

This flaw can lead to a complete breakdown of the blockchain network's security and privacy by undermining cryptographic integrity.

Impact: If `std::random_device` produces predictable values, cryptographic keys, labels, or garbled tables may become guessable, allowing attackers to decrypt data or forge outputs, ultimately compromising privacy and correctness.
Likelihood: High – Cross-platform usage (e.g., MinGW) is common, and the deterministic nature of `std::random_device` in certain environments makes it directly exploitable in cryptographic contexts.

Issue 2: Potential Buffer Overflow

The function casts a buffer to `uint*` and writes `sizeInt` integers, assuming the buffer's size is aligned to `sizeof(uint)`. If this assumption does not hold, the remainder logic could lead to memory corruption.

While it does not cause a complete network failure, it poses a significant risk of node crashes and operational disruption, leading o a high-severity vulnerability.

Impact: If the buffer size is not a multiple of `sizeof(uint)`, the remainder loop may write beyond allocated memory, leading to buffer overflows, segmentation faults, or unexpected behavior, disrupting the garbling process.
Likelihood is Moderate – While unaligned sizes may not be the default case, they are plausible (e.g., odd-byte requests). The absence of explicit validation increases the risk of misuse.

```
void Commons::generateDeviceRandomBytes(unsigned char * buffer, int
size){
uint* bufferInt = (uint*)buffer;
int sizeInt = size / sizeof(uint);
random_device rd;
for(int i = 0; i < sizeInt; i++) {
bufferInt[i] = rd();
}
if (sizeInt * sizeof(uint) < size){
int remain = size - sizeInt * sizeof(uint);
for (int i = 0; i < remain; i++){
buffer[sizeInt * sizeof(uint) + i] = static_cast<unsigned char>(rd(
));
}
}
}
```

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Severity: Critical

Recommendations

Remediation: To address the identified issues, the following improvements should be implemented:

1. Use OpenSSL for cryptographic randomness;
2. Fill the buffer sequentially, one byte at a time.

```
for (size_t i = 0; i < size; i++) {
buffer[i] = static_cast<unsigned char>(rd());
}
```

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/c1f940b7e4db7212382a057320795ec127c609e1

F-2025-8667 - Buffer Overflow Risk in Commons::convertHexToBytes - High

Description:

File location: *lib/src/commons/commons.cpp*

The function uses `sscanf` to parse two hexadecimal characters from the input string (`hexString`) and store them into a single byte in the `bytes` array. While the function checks that the `bytes` array has enough space (`size >= outputSize`), the use of `sscanf` is inherently risky because:

1. Lack of Bounds Checking in `sscanf`:

`sscanf` does not inherently check the bounds of the destination buffer (`&bytes[i]`). If the input string (`hexString`) is malformed or contains unexpected data, it could lead to undefined behavior or buffer overflow.

2. Potential for Malicious Input:

If the input string is not properly sanitized, an attacker could craft a malicious input that exploits the `sscanf` behavior to overwrite adjacent memory.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Unsafe use of `sscanf` without full bounds protection risks buffer overflow or undefined behavior (UB) if `hexString` is malformed (e.g., exceeds %2hhx expectation), potentially crashing the node or corrupting memory, disrupting garbling ops.
Likelihood: Moderate—requires malformed or malicious `hexString` (e.g., attacker-controlled input); plausible in multi-party setups or parsing errors, though size check mitigates some risk.

```
void Commons::convertHexToBytes(string hexString, unsigned char * b
ytes, int size)
{
// Ensure that the input string has an even number of characters
if (hexString.size() % 2 != 0)
{
Logger::getInstance().critical("Hex string must have an even number
of characters");
throw runtime_error("Hex string must have an even number of charact
ers");
}
int outputSize = hexString.size() / 2;
if (size < outputSize){
Logger::getInstance().critical("Output array does not have enough s
pace");
throw runtime_error("Output array does not have enough space");
}
for (size_t i = 0; i < outputSize; ++i)
{
```

```
if (sscanf(hexString.c_str() + 2 * i, "%2hhx", &bytes[i]) != 1) //
Vulnerable to buffer overflow
{
Logger::getInstance().critical("Hex string parsing error");
throw runtime_error("Hex string parsing error");
}
}
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: To enhance security and reliability, replace `sscanf` with a safer alternative that explicitly validates input and ensures proper bounds checking. Consider implementing a custom hex-parsing function that provides full contract of the parsing logic.

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/bae90ba323976c1b5e05d91b737fa16074620ec8

F-2025-8691 - Possible Memory Leaks With Context - High

Description: File: *lib/src/communication/streamCommunication.cpp*

The `clientStream` is used without being validated on the following line:

```
clientStream->WritesDone();
```

If `clientStream` is `null`, this will result in a segmentation fault.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: If clientStream is null, calling `clientStream->WritesDone()` triggers a segmentation fault, crashing the node (e.g., Garbler or Evaluator) and disrupting communication (e.g., garbled circuit transfer). *Likelihood:* High—`null` `clientStream` is plausible if initialization fails (e.g., connection error) or if reused post-destruction; common in pointer-heavy code without checks.

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: It is recommended to add a null check before using `clientStream`:

```
if (clientStream) {
clientStream->WritesDone();
Status status = clientStream->Finish();
// Handle errors...
}
```

```
void BiDirectionalStreamComm::createStream(){
// Wait for the connection to be established
grpc_connectivity_state state = channel->GetState(true);
while (state != GRPC_CHANNEL_READY) {
Logger::getInstance().info("Waiting for server (" + otherPartyName
+ ") to be ready...");
// Wait for connection to be ready or timeout (2 seconds)
if (!channel->WaitForConnected(chrono::system_clock::now() + chrono
::seconds(2))) {
Logger::getInstance().error("Failed to connect to the server (" + o
therPartyName + "). Retrying...");
}
state = channel->GetState(true); // Get the new state after waiting
}
Logger::getInstance().debug("Channel state: " + to_string(state));
// If there is an old stream, close it
if (clientStream) {
clientStream->WritesDone();
// Close the connection and check for any errors
Status status = clientStream->Finish();
if (!status.ok()) {
Logger::getInstance().error("gRPC connection closed with error: " +
status.error_message());
} else {
Logger::getInstance().info("gRPC connection closed successfully.");
}
this_thread::sleep_for(chrono::seconds(10)); // Wait for the connec
tion to close
}
bool connect = false;
while(!connect){
// Create a new context and stream
context = make_shared\<ClientContext>();
clientStream = stub->Communicate(context.get()); // Bidirectional s
treaming
Logger::getInstance().info("Client connected to " + otherPartyName
+ " on address: " + serverAddress);
connect = clientStream->Write(CommunicationMessage());
if (!connect) {
Logger::getInstance().error("Failed to connect to the server " + ot
herPartyName);
this_thread::sleep_for(chrono::milliseconds(100));
}
}
Logger::getInstance().info("Client sent init message");
}
```

Resolution:

The issue is fixed with these 2 commits:

https://github.com/soda-mpc/soda-mpc/commit/7659f3bae3d3c9a945ed6e6532b7865156d6b0b1

https://github.com/soda-mpc/soda-mpc/commit/c1f7692f1c7ee5a3cc80bfc1e33529bbb88595b0

F-2025-8692 - Potential Race Condition in waitForClient() Function - High

Description: File: *lib/src/communication/streamComm.cpp*

The function first reads `clientDisconnected` outside the `unique_lock` scope:

```
bool clientDisconnected;
{
lock_guard<mutex> lock(serviceImpl.disconnectMutex);
clientDisconnected = serviceImpl.clientDisconnected;
}
```

If `clientDisconnected` is true, it proceeds to lock again with:

```
unique_lock<mutex> lock(serviceImpl.disconnectMutex);
```

The problem is that between the time the first lock (`lock_guard`) is released and the second lock (`unique_lock`) is acquired, another thread could have already modified `clientDisconnected`. This could lead to:

- A spurious wait, where the thread waits when it shouldn't.
- A missed wake-up, where the thread doesn't wait when it should.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Race condition between `clientDisconnected` checks allow spurious waits (e.g., thread stalls unnecessarily) or missed wake-ups (e.g., thread proceeds on the stale state), potentially hanging comms or corrupting garbling data, disrupting the protocol.
Likelihood: High—multi-threaded access to `clientDisconnected` is likely in a comms service; the timing window is small but exploitable under load.

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Consider making the check and wait atomic:

```
void BiDirectionalStreamComm::waitForClient() {
unique_lock<mutex> lock(serviceImpl.disconnectMutex);
if (serviceImpl.clientDisconnected) {
Logger::getInstance().info("Server stream is closed, waiting for th
e client to connect...");
serviceImpl.disconnectCondition.wait(lock, [this]() { return !servi
ceImpl.clientDisconnected; });
}
}
```

This ensures that the check and wait condition are performed atomically, preventing any race conditions where another thread could modify clientDisconnected between the check and the wait.

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/b9c792b35cbe5f5d045249006799abfebeb281c8

F-2025-8815 - Side-Channel Vulnerability Due to Signal Bit Manipulation - High

Description: File: *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*

Non-constant time operations may expose information about signal bits, potentially leading to side-channel attacks.

This is a high-severity item, as it may cause substantial harm by risking partial privacy breaches and disruption, despite not causing a complete network breakdown.

Impact: Non-constant-time branching on `getSignalBitOf(garbledWires[startOutputIndex])` creates a timing side channel, potentially leaking signal bits (e.g., garbled wire values), compromising privacy (e.g., input inference) and weakening garbling security.
Likelihood: Moderate—requires precise timing attacks (e.g., co-located nodes or local access); plausible in multi-party or adversarial environments.

```
if (getSignalBitOf(garbledWires[startOutputIndex]) == 0){
garbledWires[lastWireIndex + 1 + 2 * i] = garbledWires[startOutputI
ndex];
*((unsigned char *)(&encryptedChunkKeys[i])) |= 1;
garbledWires[lastWireIndex + 1 + 2 * i + 1] = encryptedChunkKeys[i]
;
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Perform signal bit operations in constant time to prevent potential leakage of sensitive information through timing variations.

```
// Constant-time signal bit operations
void setSignalBitConstantTime(block& key, bool value) {
unsigned char mask = value ? 1 : 0;
unsigned char* keyBytes = (unsigned char*)&key;
keyBytes[0] = (keyBytes[0] & ~1) | mask; // Constant time operation
s
}
// In the garbling code:
unsigned char signalBit = getSignalBitOf(garbledWires[startOutputIn
dex]);
block wire0 = signalBit ? encryptedChunkKeys[i] : garbledWires[star
tOutputIndex];
block wire1 = signalBit ? garbledWires[startOutputIndex] : encrypte
dChunkKeys[i];
setSignalBitConstantTime(wire1, 1); // Ensure wire1 has signal bit
1
garbledWires[lastWireIndex + 1 + 2 * i] = wire0;
garbledWires[lastWireIndex + 1 + 2 * i + 1] = wire1;
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/a4521f95c9e062e0d7ff9f3e671c6f72062d3f38

F-2025-8818 - Risk of Integer Overflow Across Multiple Files - High

Description:

Files:

- *lib/src/crypto/circuit/garblingCircuit.cpp*
- *lib/src/crypto/circuit/booleanCircuit.cpp*
- *lib/src/crypto/circuit/circuitGarbler.cpp*
- *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolEvaluator.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolGarbler.cpp*
- *soda-mpc/opcodesTranslator/opcodesToMPC.cpp*

There is no check on the size of `numberOfInput` and/or the file size, a recurring issue across multiple files within the `circuit` folder. Malicious input files could provide excessively large values, potentially causing memory exhaustion.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Unchecked numberOfInputs from a file allow maliciously large values (e.g., 2^{31}) to resize inputs, exhausting memory and crashing the node (e.g., std::bad_alloc), disrupting garbling operations.
Likelihood: High—file-based inputs are common; lack of validation invites exploitation by crafted files in multi-party or compromised setups.

```
vector<int> GarblingCircuit::readInputsFromFile(...) {
myfile >> numberOfInputs;
inputs.resize(numberOfInputs); // Potential overflow
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Perform a bounds check on the input file to ensure the size is within acceptable limits.

```
constexpr size_t MAX_INPUTS = 1<<30; // 1 billion or whatever maxim
um is suitable
if(numberOfInputs < 0 || numberOfInputs > MAX_INPUTS)
throw std::range_error("Invalid input count");
inputs.reserve(numberOfInputs); // Safer allocation
```

Resolution: The issue is fixed with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/c9b0f364bb62865b55ce1b7f2cfc5ee5cc7161dc

https://github.com/soda-mpc/soda-mpc/commit/2b48ceedf458c2c1797818c2fee1b4cfb2845964

https://github.com/soda-mpc/soda-mpc/commit/8aef75240ffbe3d52b3d6953ffdf85e41bdd897a

https://github.com/soda-mpc/soda-mpc/commit/5b876ae3907957881206cfb8d8b715dcef0d5cc1

https://github.com/soda-mpc/soda-mpc/commit/08f9f311823c2a6f109f7b035ec9669b56001a06

https://github.com/soda-mpc/soda-mpc/commit/33da7232aa115f1a32fee5c03affd5b25b498389

https://github.com/soda-mpc/soda-mpc/commit/d37f870d81cd062b60d43f8b14e4ff5b31649d4e

https://github.com/soda-mpc/soda-mpc/commit/7ef92413bf4ab7cb9e2d04eaf0fd47dbef12239d

F-2025-8827 - Non-Constant Time Operations in Signal Bit Comparison - High

Description:	File: *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*

The code contains branches that create non-constant timing side channels during signal bit comparison, specifically:

```
if (wire0SignalBit == 0) {
tempK0 = _mm_xor_si128(encryptedKeys[0], keys2[0]);
}
else {
tempK0 = _mm_xor_si128(_mm_xor_si128(encryptedKeys[0], keys2[0]),
garbledTables[2 * nonXorIndex]);
}
```

The conditional branching based on `wire0SignalBit` introduces different execution paths, which can lead to variations in execution time based on the input. This timing discrepancy could potentially leak information about the signal bit, compromising security by creating a side channel.

This is a high-severity item, as it may cause substantial harm by risking partial privacy breaches and disruption, despite not causing a complete network breakdown.

Impact: Non-constant-time branching on wire0SignalBit creates a timing side channel, potentially leaking garbled circuit signal bits (e.g., wire labels), compromising privacy (e.g., input inference), and weakening protocol security.
Likelihood: Moderate—requires precise timing attacks (e.g., via co-located nodes or local access); plausible in multi-party or adversarial settings.

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	3/5
Severity:	High

Recommendations

Remediation:	Modify this operation to ensure constant-time execution.

```
// Constant-time selection
block mask = _mm_set1_epi8(wire0SignalBit ? 0xFF : 0x00);
block tableValue = _mm_and_si128(mask, garbledTables[2 * nonXorInde
x]);
tempK0 = _mm_xor_si128(_mm_xor_si128(encryptedKeys[0], keys2[0]), t
ableValue);
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/9692e76f88c7c26d06628b254949dce5529a37e8

F-2025-8884 - Absence of Message Size Validation in RSA Encryption - High

Description:

Files:

- *lib/src/crypto/encryption/assymmetricEncryption.cpp*
- *lib/src/crypto/encryption/cryptoppRsa.cpp*

The RSA encryption algorithm can only encrypt small messages at a time, as the message must be smaller than the key size minus the OAEP padding overhead.

For a 2048-bit key, the size is 256 bytes, and with SHA256 used for padding (which is 32 bytes), the maximum size for the plaintext is calculated as:

```
max_plaintext = key_size_in_bytes − 2 * sha_length_in_bytes − 2
```

In this case:

- The key size is 2048 bits (or 256 bytes).
- The hash size is SHA256 (or 32 bytes).

Thus:

```
max_plaintext = 256 − 2 * 32 − 2 = 190 bytes
```

It is recommended that the function checks the length of the plaintext before attempting encryption to ensure it does not exceed this limit.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Encrypting a plaintext larger than 190 bytes (for 2048-bit RSA with OAEP/SHA256) in `OpenSSL_RSAOAEP::encrypt` or `CryptoPP_RSAOAEP::encrypt` triggers buffer overruns or encryption failures, potentially crashing the node (e.g., segfault) or producing corrupt ciphertext, disrupting garbled circuit comms.
Likelihood: High—plaintext sizes exceeding 190 bytes are plausible (e.g., circuit data, keys); lack of checks ensures failure under common conditions.

```
void OpenSSL_RSAOAEP::encrypt(vector<unsigned char> & plaintext, ve
ctor<unsigned char> & ciphertext) {
// No check for maximum message size with OAEP padding
size_t outSize = ciphertext.size();
if (EVP_PKEY_encrypt(ctxEncrypt, ciphertext.data(), &outSize, plain
text.data(), plaintext.size()) <= 0) {
```

```
throw runtime_error("Error encrypting data");
}
}


void CryptoPP_RSAOAEP::encrypt(...) {
// Check the size of the public key
if (publicKeyDER.size() != RSA_PUBLIC_KEY_SIZE){
Logger::getInstance().critical("Public key size is not correct");
throw invalid_argument("Public key size is not correct");
}
// No check for plaintext size against RSA key size
…
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Consider adding a plaintext length check:

```
void OpenSSL_RSAOAEP::encrypt(vector<unsigned char> & plaintext, ve
ctor<unsigned char> & ciphertext) {
// OAEP padding overhead is 66 bytes for SHA-256
size_t maxSize = EVP_PKEY_size(pkey) - 66;
if (plaintext.size() > maxSize) {
throw runtime_error("Plaintext too large for RSA-OAEP encryption");
}

size_t outSize = ciphertext.size();
if (EVP_PKEY_encrypt(...) <= 0) {
throw runtime_error("Error encrypting data");
}
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/be20fedb3038c4b0465dfd29e3b1e8efec97e38b

F-2025-8938 - Insufficient Input Validation - High

Description: File: *lib/src/crypto/hmac/hmac.cpp*

The `setKey` method lacks validation for an empty key and does not check for an acceptable key size.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Passing an empty or invalid-sized key to EVP_MAC_init may crash the node (e.g., OpenSSL segfault) or produce weak HMACs (e.g., empty key accepted), compromising garbling integrity and disrupting protocol security.
Likelihood: Moderate—empty or malformed keys are plausible from caller errors or malicious inputs; lack of validation ensures failure under these conditions.

```
void OpenSSL_HMAC_SHA256::setKey(vector<unsigned char> & key){
if (!mac_ctx || !mac) {
Logger::getInstance().critical("MAC context or MAC not initialized"
);
throw runtime_error("MAC context or MAC not initialized");
}
// Set the key for Hmac
OSSL_PARAM params[] = {
OSSL_PARAM_construct_utf8_string("digest", const_cast<char*>("SHA25
6"), 0),
OSSL_PARAM_construct_end()
};
if (EVP_MAC_init(mac_ctx, key.data(), key.size(), params) <= 0) {
EVP_MAC_CTX_free(mac_ctx);
EVP_MAC_free(mac);
Logger::getInstance().critical("Failed to initialize MAC context");
throw runtime_error("Failed to initialize MAC context");
}
isInitialized = true;
}
```

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	3/5
Severity:	High

Recommendations

Remediation: Validate that the key is not empty and is within the acceptable size bounds.

```
void setKey(const std::vector<unsigned char>& key) {
if (key.empty() || key.out_of_bound()) {
throw std::invalid_argument("Key must not be empty");
}
// Proceed with MAC initialization...
}
```

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/de48e94dcc16b6a4e3b48c7a9c37febe35ece141

F-2025-8939 - Absence of Secure Memory Wiping in Destructor - High

Description: File: *lib/src/crypto/hmac/hmac.cpp*

The destructor frees the memory but does not wipe the sensitive information within it.

```
OpenSSL_HMAC_SHA256::~OpenSSL_HMAC_SHA256() {
if (mac_ctx) {
EVP_MAC_CTX_free(mac_ctx);
}
if (mac) {
EVP_MAC_free(mac);
}
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Use `OpenSSL_cleanse()` to wipe sensitive memory information.

```
OpenSSL_HMAC_SHA256::~OpenSSL_HMAC_SHA256() {
if (mac_ctx) {
EVP_MAC_CTX_reset(mac_ctx);
OPENSSL_cleanse(mac_ctx, sizeof(*mac_ctx));
EVP_MAC_CTX_free(mac_ctx);
mac_ctx = nullptr;
}
if (mac) {
EVP_MAC_free(mac);
mac = nullptr;
}
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/7b109e4e3139c02ef10e29022c4492409c2a9f85

F-2025-8941 - Potential Underflow in OpenSSLHash::hash() - High

Description: File: *lib/src/crypto/hash/hash.cpp*

The `hash()` method takes an `int` size without validating its non-negativity. A negative size could result in an underflow when casting to `size_t`.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: A negative size underflows when cast to size_t (e.g., -1 becomes 2^{64}-1), causing hash to process massive or invalid memory regions, potentially crashing the node (e.g., segfault) or producing corrupt hashes, disrupting garbling integrity.
Likelihood: Moderate—requires negative size input (e.g., caller error or malicious input); plausible in unvalidated multi-party contexts.

```
bool OpenSSLHash::hash(unsigned char* bytesToHash, int size, unsign
ed char* hashOutput) {
size_t remainingDataSize = size;
unsigned char* currentData = bytesToHash;
// Call EVP_DigestUpdate for each block of data
while (remainingDataSize > 0) {
// Determine the size of the current block of data
size_t blockSizeToProcess = (remainingDataSize > blockSize) ? block
Size : remainingDataSize;

if (!update(currentData, blockSizeToProcess)) {
Logger::getInstance().critical("Error: Failed to update hash contex
t");
return false;
}
// Move to the next block of data
currentData += blockSizeToProcess;
remainingDataSize -= blockSizeToProcess;
}
// Finalize the hash computation and store the result in hashOutput
if (!finalize(hashOutput)) {
Logger::getInstance().critical("Error: Failed to finalize hash hash
");
return false;
}
// Initialize the context for the next hash computation
if (!EVP_DigestInit_ex(ctx, md, nullptr)){
Logger::getInstance().critical("Error: Failed to initialize hash co
ntext");
return false;
}
return true;
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate:	4/5
Severity:	High

Recommendations

Remediation:	Validate that the size is non-negative before proceeding with any other operations.
Resolution:	The issue is fixed with this commit: https://github.com/soda-mpc/soda-mpc/commit/e8e8b1e57ecc46d56f4a7dd5730d32415a25df87

F-2025-8946 - Absence of Null Check Following Memory Allocation - High

Description: File: *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolGarbler.cpp:103*

The code does not verify whether memory was successfully allocated:

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption despite not causing a complete network breakdown.

Impact: Failing to check if `_aligned_malloc` returns `null` (e.g., due to memory exhaustion) leads to dereferencing a null pointer in subsequent operations, crashing the node (e.g., segfault), and halting garbling processes.
Likelihood: Moderate—memory exhaustion is plausible under load (e.g., large `numberOfInputs`); depends on system resources but lack of check ensures failure.

```
block* labelsAuthentication = (block *)_aligned_malloc(numberOfInpu
ts * 2 * SIZE_OF_BLOCK, SIZE_OF_BLOCK);
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Implement a null check to verify successful memory allocation.

```
block* labelsAuthentication = static_cast<block*>(_aligned_malloc(n
umberOfInputs * 2 * SIZE_OF_BLOCK, SIZE_OF_BLOCK));
if (labelsAuthentication == nullptr) {
Logger::getInstance().critical("labelsAuthentication could not be a
llocated");
throw std::bad_alloc();
}
```

Resolution: The issue is fixed with this commit::

https://github.com/soda-mpc/soda-mpc/commit/87d892914245fd48de09d3443e39205da631aeb8

F-2025-8947 - Memory Management Issues in Garbler::generateBatch() - High

Description:

File: *opcodestranslator/garbler.cpp*

Several issues identified in `Garbler::generateBatch()`:

Issue 1: Manual Memory Management Risks The method allocates an array of pointers (`new unsigned char*[3]`) and later deletes individual elements (`garblerOutputBufs[0]`, `[1]`, `[2]`) before deleting the array itself (`delete garblerOutputBufs`). If an exception occurs between allocation and deletion, some memory may not be freed, leading to memory leaks.

This is a medium-severity item, as it could negatively affect the protocol by causing memory leaks under specific conditions, but it's not catastrophic.

Impact: An exception (e.g., from `garbler.runRound()`) before deletions leaves `garblerOutputBufs[0-2]` and the array unfreed, leaking memory and potentially degrading node performance over time.
Likelihood: Moderate—exceptions are plausible (e.g., garbling errors), but require specific failures; good-faith operation bounds frequency.

Issue 2: Lack of Null Pointer Check After `new` **Allocation** On line 98, the code allocates memory with `new unsigned char*[3]` but does not verify whether the allocation succeeded or failed.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption despite not causing a complete network breakdown.

Impact: If new unsigned `char*[3]` fails (e.g., out of memory) and returns `null`, dereferencing `garblerOutputBufs` (e.g., in runRound) causes a segfault, crashing the node and halting garbling.
Likelihood: Moderate—memory exhaustion is possible under load (e.g., large batches); depends on system resources.

Issue 3: Early Exit on Exception Handling On lines 134–137, if `garbler.runRound()` throws an exception, the loop exits prematurely, preventing the execution of `delete[]` for `garblerOutputBufs`.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: An exception in `garbler.runRound()` exits the loop, skipping `delete[] garblerOutputBufs`, leaking memory and risking node instability

or crash (e.g., exhaustion), disrupting garbling.
Likelihood: High—garbling errors (e.g., invalid circuit, memory failure) are plausible in production; no exception handling amplifies risk.

Issue 4: Improper Exception Handling with `dynamic_cast` On line 141, `dynamic_cast<ProtocolGarblerData*>(data)` is used without proper exception handling. If the cast fails, it throws a `std::bad_cast` exception, which may not be caught by the existing null check on line 142.

```
auto *pGarblerData = dynamic_cast<ProtocolGarblerData*>(data);
if (pGarblerData == nullptr) {
Logger::getInstance().critical("The given input must be an instance
of ProtocolGarblerData");
throw invalid_argument("The given input must be an instance of Prot
ocolGarblerData");
}
```

This is a medium-severity item, as it could negatively affect the protocol by risking unhandled exceptions under specific conditions, but it's not catastrophic.

Impact: `dynamic_cast` throws `std::bad_cast` (not `null`) if data isn't `ProtocolGarblerData*` when casting from a polymorphic base with RTTI, crashing the node if uncaught, disrupting garbling.
Likelihood: Low—requires misconfigured data (e.g., wrong type passed); rare in good-faith setup but possible with bugs.

The issues described above can lead to severe memory management issues, potential crashes, and undefined behavior, especially in production environments. They require prompt attention to ensure the robustness and stability of the system.

```
void Garbler::generateBatch(BcNames & bcName, uint64_t & id, vector
<unsigned char> & batchGarbledTables, vector<unsigned char> & share
s0,
vector<unsigned char> & shares1, vector<unsigned char> & signatureG
arbledTables,
vector<unsigned char> & signatureShares0, vector<unsigned char> & s
ignatureShares1){
auto bc = bcHolder.getCircuit(bcName);
int64_t capacity = util.getCircuitsInBatchCapacity(bcName);
if (capacity == 0){ // Skip the circuit if the capacity is 0
return;
}
vector<unsigned char> batchData0;
vector<unsigned char> batchData1;
unsigned char** garblerOutputBufs = new unsigned char*[3];
int size = 0, gtSize;
// Run the garbler to create batch of garbling data
for (int16_t j=0; j<capacity; j++){
// Generate a unique message for each circuit in the batch
vector<unsigned char> message(CIRCUIT_ID_SIZE + BATCH_ID_SIZE + CIR
CUIT_IN_BATCH_SIZE + AES_KEY_SIZE + 1);
int16_t bcName16 = static_cast<int16_t>(bcName);
memcpy(message.data(), &bcName16, CIRCUIT_ID_SIZE);
memcpy(message.data() + CIRCUIT_ID_SIZE, &id, BATCH_ID_SIZE);
memcpy(message.data() + CIRCUIT_ID_SIZE + BATCH_ID_SIZE, &j, CIRCUI
T_IN_BATCH_SIZE);
memcpy(message.data() + CIRCUIT_ID_SIZE + BATCH_ID_SIZE + CIRCUIT_I
N_BATCH_SIZE, garblerSeed.data(), AES_KEY_SIZE);
// Generate the seeds for the garbled circuit
// Each seed is generated by computing HMAC on the message with a d
ifferent value in the last byte.
// This ensures unique message to the hmac, which will generate a u
nique seed.
block evaluator0Seed, evaluator1Seed;
// Generate seed for evaluator 0
message[CIRCUIT_ID_SIZE + BATCH_ID_SIZE + CIRCUIT_IN_BATCH_SIZE + A
ES_KEY_SIZE] = 0;
vector<unsigned char> seed = hmacForGarbling.hmac(message);
memcpy(&evaluator0Seed, seed.data(), sizeof(block));
// Generate seed for evaluator 1
message[CIRCUIT_ID_SIZE + BATCH_ID_SIZE + CIRCUIT_IN_BATCH_SIZE + A
ES_KEY_SIZE] = 1;
seed = hmacForGarbling.hmac(message);
memcpy(&evaluator1Seed, seed.data(), sizeof(block));
// Generate one more seed for the shares generation
message[CIRCUIT_ID_SIZE + BATCH_ID_SIZE + CIRCUIT_IN_BATCH_SIZE + A
ES_KEY_SIZE] = 2;
seed = hmacForGarbling.hmac(message);
block sharesSeed;
memcpy(&sharesSeed, seed.data(), sizeof(block));
// Run the garbler to create the garbled circuit using the generate
d seeds
TwoEvaluatorsProtocolGarbler garbler(&bc, 3, 1, evaluator0ItMacKey,
evaluator1ItMacKey, evaluator0Seed, evaluator1Seed, sharesSeed);
while (!garbler.isFinished()){
// The garbler does not take any inputs
garbler.runRound(nullptr, 0, garblerOutputBufs, &size);
}

// Get the garbled table size
auto data = garbler.getDataPtr();
auto *pGarblerData = dynamic_cast<ProtocolGarblerData*>(data);
if (pGarblerData == nullptr) {
Logger::getInstance().critical("The given input must be an instance
of ProtocolGarblerData");
throw invalid_argument("The given input must be an instance of Prot
ocolGarblerData");
}
gtSize = pGarblerData->garbler->getGarbledTableSize();
// Allocate memory for the batch in the first time
if (batchData0.size() == 0){
batchGarbledTables.resize(gtSize * capacity);
batchData0.resize(size * capacity);
batchData1.resize(size * capacity);
}
// Copy the garbling data to the batch
memcpy(batchGarbledTables.data() + j*gtSize, garblerOutputBufs[0],
gtSize); // NOLINT The linter incorrectly raises alarms regarding d
eallocated memory; instruct it to disregard.
// Copy the shares of the permutation bits and the seed
memcpy(batchData0.data() + j*size, garblerOutputBufs[1], size);
memcpy(batchData1.data() + j*size, garblerOutputBufs[2], size);
delete [] garblerOutputBufs[0];
delete [] garblerOutputBufs[1];
delete [] garblerOutputBufs[2];
}
// Generate the pseudo random IV by encrypting the circuit ID, the
batch ID and the party ID with the garbler AES key
vector<unsigned char> message(AES_KEY_SIZE + 1);
int64_t bcName64 = static_cast<int64_t>(bcName);
memcpy(message.data(), &bcName64, AES_KEY_SIZE/2);
memcpy(message.data() + AES_KEY_SIZE/2, &id, AES_KEY_SIZE/2);
// Generate the IVs for the encryption of the shares
// The IVs are generated by the HMAC of the message with different
values in the last byte.
message[AES_KEY_SIZE] = 0;
vector<unsigned char> iv0 = hmacForEncryption.hmac(message);
iv0.resize(AES_KEY_SIZE); // Get the first 128 bits as IV to the AE
S encryption
message[AES_KEY_SIZE] = 1;
vector<unsigned char> iv1 = hmacForEncryption.hmac(message);
iv1.resize(AES_KEY_SIZE); // Get the first 128 bits as IV to the AE
S encryption

// Encrypt the shares
encryptData(batchData0, iv0, 0, shares0);
encryptData(batchData1, iv1, 1, shares1);
// Generate signatures on the batch data
signBatch(batchGarbledTables, bcName, id, signatureGarbledTables);
signBatch(shares0, bcName, id, signatureShares0);
signBatch(shares1, bcName, id, signatureShares1);
```

```
delete [] garblerOutputBufs;
}
```

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	4/5
Severity:	High

Recommendations

Remediation:

To address the identified issues in the `Garbler::generateBatch()` method, the following actions are recommended:

1. Replace manual memory management with RAII (e.g., using smart pointers) to ensure memory is automatically freed and prevent potential memory leaks in the presence of exceptions.
2. Add null pointer checks after dynamic memory allocation (e.g., `new unsigned char*[3]`) to ensure the allocation succeeded before using the allocated memory.
3. Implement RAII or use `try-catch` blocks to ensure proper cleanup of resources, even if an exception occurs.
4. Enclose the `dynamic_cast` in a `try-catch` block to properly handle `std::bad_cast` exceptions, preventing crashes due to invalid type casts.

Implementing these changes will improve memory safety, stability, and robustness of the garbler process.

Resolution:

The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/a477465610e212437e21d27fcd1afacad95e7b1d

F-2025-8948 - Race Condition Due to Unsynchronized Access to exitThreads - High

Description: File: *opcodesTranslator/garblingManager.cpp*

The `exitThreads` variable is a non-atomic `bool` that is accessed by multiple threads without any synchronization mechanisms (such as a mutex or atomic). As a result, concurrent read and write operations on `exitThreads` can lead to undefined behavior, potentially causing data races or inconsistent state.

```
void GarblingManager::startThreads(){
exitThreads = false;
// Start the thread responsible for retrieving garbling data from t
he database.
gcRetrievalThread = thread(&GarblingManager::gcRetrieval, this);
// Start the thread responsible for notifying the GCManager of the
current state.
notifyGCMThread = thread(&GarblingManager::notifyGCManager, this);
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Ensure thread-safe access to `exitThreads`.

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/454db5b07a9f013ce82d780596296a159e17f20c

F-2025-8951 - Non Atomic And Non Join Safety In Destructor - High

Description:

File: *opcodesTranslator/garblingManager.cpp*

The destructor code below contains two significant issues:

Issue 1: Non-atomic `exitThreads` variable

The `exitThreads` variable is not declared as atomic, which allows for concurrent reads and writes without synchronization. This can result in undefined behavior, potentially leading to node crashes or disruptions. While this does not directly cause a complete breakdown of the network, the risks associated with this issue are substantial.

Impact: Non-atomic `exitThreads` leads to undefined behavior (UB) from concurrent reads/writes (e.g., destructor vs. worker threads), potentially crashing the node (e.g., segfault) or causing hangs (e.g., threads miss true), disrupting garbling.
Likelihood: High—multi-threaded access during shutdown is common; lack of sync invites data races.

Issue 2: Lack of thread join safety

The code does not check whether a thread is joinable before attempting to join it. This omission can lead to exceptions if the thread has already been joined or detached. Ensuring that the thread is joinable before calling `join()` would prevent such exceptions and improve the robustness of the code.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Joining non-joinable threads (e.g., already joined or unstarted) throws `std::system_error`, crashing the node if uncaught, disrupting garbling cleanup and Evaluator comms.
Likelihood: Moderate—depends on thread state (e.g., prior join or failure to start); plausible in dynamic shutdown scenarios.

```
GarblingManager::~GarblingManager(){
exitThreads = true;

gcRetrievalThread.join();
notifyGCMThread.join();
}
```

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	4/5
Severity:	High

Recommendations

Remediation: To resolve the issues:

1. **Use atomic variables** for shared data like `exitThreads` to prevent race conditions.
2. **Check thread joinability** before calling `join()` to avoid exceptions from already joined or detached threads.

Resolution: The issue is fixed with these 2 commits:

https://github.com/soda-mpc/soda-mpc/commit/454db5b07a9f013ce82d780596296a159e17f20c

https://github.com/soda-mpc/soda-mpc/commit/4308bc6da7be07bc63512e942a2e93e0c28008f6

F-2025-8968 - Unsafe Exception Handling in GCManager::refillCapacities - High

Description: File: *opcodesTranslator/gcManager.cpp*

In `GCManager::refillCapacities`, lines 231-234, the catch block rethrows the exception using `throw e;`, allowing it to escape the thread's function. If uncaught at a higher level, this will trigger `std::terminate` and abruptly terminate the program.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Rethrowing the exception in `refillCapacities` escapes the thread (e.g., `refillCapacitiesThread`), triggering `std::terminate` if uncaught higher up, crashing the node and halting garbling operations. *Likelihood:* Moderate—depends on exceptions occurring (e.g., garbler data errors) and lack of outer try-catch; plausible under failure conditions in production.

```
try {
// Process queue
} catch (runtime_error & e) {
Logger::getInstance().critical("Error while getting garbler data: "
+ string(e.what()));
throw e;
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Handle the exception within the thread, signal other threads to exit.

```
try {
// Process queue
} catch (const std::runtime_error& e) {
Logger::getInstance().critical("Critical error in refillCapacities:
" + std::string(e.what()));
std::lock_guard<std::mutex> lock(queueMutex);
exitProgram = true; // Signal thread to exit
```

```
    cv.notify_one();
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/dfe38944a9d15aae52f5a37929ac79f0a67a2035

F-2025-8970 - Unsafe Destructor Implementation in GCManager - High

Description: File: *opcodesTranslator/gcManager.cpp*

In the destructor, calling `join()` on `refillCapacitiesThread` when it isn't joinable (e.g., not started or already joined) results in undefined behavior.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Calling `join()` on a non-joinable *refillCapacitiesThread* (e.g., unstarted or already joined) throws `std::system_error`, crashing the node if uncaught, disrupting garbling cleanup and protocol operation.
Likelihood: Moderate—depends on thread state (e.g., not started due to init failure or joined elsewhere); plausible in dynamic or error-prone scenarios.

```
GCManager::~GCManager() {
lock_guard<mutex> lock(queueMutex);
exitProgram = true;
cv.notify_one();
refillCapacitiesThread.join();
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Check `joinable()` before calling `join()` on `refillCapacitiesThread`.

```
GCManager::~GCManager() {
{
std::lock_guard<std::mutex> lock(queueMutex);
exitProgram = true;
}
cv.notify_one();
if (refillCapacitiesThread.joinable()) {
refillCapacitiesThread.join();
```

```
  }
  }
```

Resolution: The issue is resolved with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/454db5b07a9f013ce82d780596296a159e17f20c

https://github.com/soda-mpc/soda-mpc/commit/4308bc6da7be07bc63512e942a2e93e0c28008f6

https://github.com/soda-mpc/soda-mpc/commit/244100f0ff6ae7ba0c3d2fcbe48524102dbac2d3

F-2025-9031 - Potential Memory Leak in GarblerServer - High

Description: File: *opCodesTranslator/grpc/garblerServer.hpp*

There is no mechanism in place to properly shut down the `GarblerServer` in their respective classes.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Without proper shutdown, `GarblerServer` leaves resources (e.g., gRPC server, threads) dangling, risking crashes (e.g., socket exhaustion) or data loss (e.g., incomplete circuit batches), disrupting protocol execution.
Likelihood: High - shutdowns occur in normal operations (e.g., restarts, updates); lack of cleanup is a common oversight in server designs.

```
class GarblerServer final : public GarblerService::Service {
private:
int garblerID; // The ID of the garbler
unique_ptr<Server> server; // The gRPC server
unique_ptr<Garbler> garbler; // The garbler instance that generates
the garbled circuits
OpcodesUtil util;
/**
* Starts the server and waits for incoming requests.
*/
void runServer(string commFile);
/**
* Checks if the given circuit name is valid.
*/
bool checkCircuitName(BcNames bcName);
public:
/**
* Constructor for the GarblerServer.
* Initializes the gRPC server and the garbler instance.
*/
GarblerServer(int garblerID, string commFile);
/**
* Function that handles the ProduceBatch RPC.
* ProduceBatch RPC is used to call the garbler to generate the requ
ested
* batch and get the generated values in the response.
*/
Status ProduceBatch(ServerContext* context, const ProduceRequest* r
equest, ProduceResponse* reply) override;
};
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Add a shutdown mechanism to allow graceful shutdown for both garbler and gRPC server.

```
// Add to class definition
private:
unique_ptr<Garbler> garbler;

public:
// Add destructor or shutdown method
void shutdown() {
if (garbler) {
garbler->Shutdown();
}
}

~GarblerServer() {
shutdown();
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/5be69f169b0f132092f25de96374829dc4939bfa

F-2025-9173 - Resource Leak in setServerAddress Function - High

Description: File: *lib/src/communication/streamComm.cpp:14*

In `BiDirectionalStreamComm::setServerAddress`, calling the method multiple times overwrites the existing `serverThread` without joining, leading to a thread resource leak.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Overwriting `serverThread` leaks resources (e.g., threads), potentially crashing the node via exhaustion (e.g., thread limits) or causing undefined behavior (UB) if the old thread accesses this, disrupting garbling comms.
Likelihood: High—multiple calls plausible in reconfigs or retries; common threading error in dynamic setups.

```
void BiDirectionalStreamComm::setServerAddress(bool isServer, strin
g serverAddress, string otherPartyName){
this->serverAddress = serverAddress;
this->otherPartyName = otherPartyName;
this->role = isServer ? SERVER_ROLE : CLIENT_ROLE;
if (role == SERVER_ROLE) {
// Start the server on a separate thread to handle incoming request
s
serverThread = thread(&BiDirectionalStreamComm::startServer, this,
serverAddress);
} else {
// Start the client to connect to the server
startClient(serverAddress);
}
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Join the thread before `setServerAddress`:

```
void BiDirectionalStreamComm::setServerAddress(bool isServer, strin
g serverAddress, string otherPartyName) {
if (serverThread.joinable()) {
```

```
serverThread.join(); // Clean up previous thread
}
this->serverAddress = serverAddress;
this->otherPartyName = otherPartyName;
this->role = isServer ? SERVER_ROLE : CLIENT_ROLE;
if (role == SERVER_ROLE) {
serverThread = thread(&BiDirectionalStreamComm::startServer, this,
serverAddress);
} else {
startClient(serverAddress);
}
}
```

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/d90f9f53e35916ddf9e95f81bf39c2b2572eac51

F-2025-9174 - Incomplete Cleanup in Destructor - High

Description: File: *lib/src/communication/streamComm.cpp:27-56*

In the client's destructor, `clientStream->Finish()` is not called, potentially leaving the stream open. For the server, `serverStream` is not explicitly cleaned up, which could result in incomplete synchronization with `Communicate` during shutdown.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Omitting `clientStream->Finish()` leaves the gRPC stream open, potentially hanging the client node (e.g., resource leaks) or causing incomplete comms (e.g., garbled tables not fully sent), disrupting protocol execution.
Likelihood: High—destructor runs on every shutdown; missing cleanup is a common gRPC error under load or errors.

```
BiDirectionalStreamComm::~BiDirectionalStreamComm(){
Logger::getInstance().info("Closing gRPC connection");
// Client side
if (role == CLIENT_ROLE) {
clientStream->WritesDone();
Logger::getInstance().info("Writes done");

// Server side
} else if (role == SERVER_ROLE) {
// Notify the waiting server to exit
{
lock_guard<mutex> lock(serviceImpl.exitMutex);
serviceImpl.exitFlag = true;
}
serviceImpl.exitCondition.notify_one();
if (server) {
server->Shutdown();
}
if (serverThread.joinable()){
serverThread.join();
}
} else {
Logger::getInstance().error("Role is not initialized");
}
Logger::getInstance().info("gRPC connection closed");
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Finish the client stream by calling `clientStream->Finish()` and ensure the server stream is properly shut down and cleaned up.

```
BiDirectionalStreamComm::~BiDirectionalStreamComm() {
Logger::getInstance().info("Closing gRPC connection");
if (role == CLIENT_ROLE) {
if (clientStream) {
clientStream->WritesDone();
Status status = clientStream->Finish();
if (!status.ok()) {
Logger::getInstance().error("Client stream finish failed: " + statu
s.error_message());
}
}
} else if (role == SERVER_ROLE) {
{
std::lock_guard<std::mutex> lock(serviceImpl.exitMutex);
serviceImpl.exitFlag = true;
}
serviceImpl.exitCondition.notify_one();
if (server) {
server->Shutdown();
server->Wait(); // Ensure server fully stops
}
if (serverThread.joinable()) {
serverThread.join();
}
}
Logger::getInstance().info("gRPC connection closed");
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/7e25b4e7b40c5c60a43a2a8f8e1df913735e6ecb

F-2025-9195 - Exposure of Internal State through Raw Pointers - High

Description: File: *lib/include/crypto/circuit/garblingCircuit.hpp:84,94*

Returning raw pointers to `garbledTables` and `bc` allows external code to modify the internal state directly. Since `garbledTables` contain sensitive cryptographic data, unintended modification could compromise security or correctness.

This is a high-severity item, as it may cause substantial harm by risking node crashes and disruption, despite not causing a complete network breakdown.

Impact: Returning raw pointers (`garbledTables`, `bc`) allows external code to modify sensitive cryptographic data (e.g., garbled tables) or circuit structure, potentially corrupting outputs or crashing the node (e.g., invalid memory access), disrupting garbling integrity.
Likelihood: Moderate—depends on external misuse (e.g., buggy or malicious caller); plausible in multi-party or poorly isolated codebases.

```
block* getGarbledTables() { return garbledTables; };
...
BooleanCircuit* getBooleanCircuit() { return bc; }
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: Return either copies or const references to ensure read-only access.

```
const block* getGarbledTables() const { return garbledTables; }; //
Read-only pointer
...
const BooleanCircuit* getBooleanCircuit() const { return bc; }; //
Read-only pointer
```

Resolution: The issue is solved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/a920b78bb32f05448e4b1699cf85ce9920814b9c

F-2025-8669 - Resource Cleanup Failure on Exception During Allocation - Medium

Description:

File locations:

- *lib/src/sommons/commons.cpp*
- *lib/src/crypto/circuit/circuitEvaluator.cpp*
- *lib/src/crypto/circuit/circuitGarbler.cpp*
- *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolEvaluator.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolGarbler.cpp*
- *opcodesTranslator/opcodesToMPC.cpp*

The issue is demonstrated in the *lib/src/commons/commons.cpp*, but similar problems can be found in all of the listed locations.

Specifically, in the `getPseudoRandomValues` function, memory is allocated for `indexArray` using `_aligned_malloc`. If the allocation for `encryptedChunkKeys` fails, `indexArray` is freed with `_aligned_free`, and a `bad_alloc` exception is thrown.

However, if any other exception is thrown after both allocations succeed (e.g., during the rest of the function's execution), neither `indexArray` nor `encryptedChunkKeys` will be freed, leading to a memory leak. Possible exceptions that remain unhandled include:

- Exceptions in `AES_set_encrypt_key`
- Exceptions in `AES_ecb_encrypt_chunk_in_out`
- Exceptions in `memset` (although unlikely, it is still advisable to handle it).

```
block* Commons::getPseudoRandomValues(block* seed, int numberOfRand
omValues){
// ... existing code ...
block* indexArray = (block *)_aligned_malloc(SIZE_OF_BLOCK * number
OfRandomValues, SIZE_OF_BLOCK);
if (indexArray == nullptr) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc();
}

block* encryptedChunkKeys = (block *)_aligned_malloc(SIZE_OF_BLOCK
* numberOfRandomValues, SIZE_OF_BLOCK);
if (encryptedChunkKeys == nullptr) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
_aligned_free(indexArray); // Need to free previously allocated mem
ory
throw bad_alloc();
}
// ... rest of code ...
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Severity: Medium

Recommendations

Remediation: Consider using RAII (Resource Acquisition Is Initialization) principles to manage memory and other resources more safely. By leveraging RAII, resources are automatically released when they go out of scope, reducing the risk of leaks even in the presence of exceptions.

```
block* Commons::getPseudoRandomValues(block* seed, int numberOfRand
omValues){
// Custom deleter for aligned memory
struct AlignedDeleter {
void operator()(block* p) { _aligned_free(p); }
};
// Use smart pointers to handle cleanup automatically
unique_ptr<block, AlignedDeleter> indexArray(
(block*)_aligned_malloc(SIZE_OF_BLOCK * numberOfRandomValues, SIZE_
OF_BLOCK)
);
if (!indexArray) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc();
}
unique_ptr<block, AlignedDeleter> encryptedChunkKeys(
(block*)_aligned_malloc(SIZE_OF_BLOCK * numberOfRandomValues, SIZE_
OF_BLOCK)
);
if (!encryptedChunkKeys) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
throw bad_alloc();
}
// Initialize AES key
AES_KEY aesSeedKey;
if (AES_set_encrypt_key((const unsigned char *)seed, 128, &aesSeedK
ey) != 0) {
Logger::getInstance().critical("AES key setup failed");
throw runtime_error("AES key setup failed");
}
// Set up index array
for (int i = 0; i < numberOfRandomValues; i++) {
indexArray.get()[i] = _mm_set_epi32(0, 0, 0, i);
}
memset(encryptedChunkKeys.get(), 0, SIZE_OF_BLOCK * numberOfRandomV
alues);
// Perform encryption
try {
AES_ecb_encrypt_chunk_in_out(indexArray.get(), encryptedChunkKeys.g
et(),
numberOfRandomValues, &aesSeedKey);
```

```
  } catch (...) {
  Logger::getInstance().critical("AES encryption failed");
  throw runtime_error("AES encryption failed");
  }
  // Transfer ownership to caller
  block* result = encryptedChunkKeys.release();
  return result;
  }
```

Resolution: The issue is fixed with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/86fcda96f944b8ecdd78dfcaab1bc0b6dda0cf01

https://github.com/soda-mpc/soda-mpc/commit/625f528e3b59e304327316456b40a9c5c56841cd

F-2025-8698 - Memory and Exception Handling Issues in BooleanCircuit Constructor - Medium

Description: File: *lib/src/crypto/circuit/booleanCircuit.cpp*

The code below exhibits two issues:

1. The constructor lacks exception handling for the case where `readCircuitFromFile` fails.
2. There are no checks for maximum file size, maximum wire index, or other relevant bounds (e.g., `numberOfGate`), which could result in a buffer overflow. It is advisable to implement checks for these values to mitigate such risks.

```
BooleanCircuit::BooleanCircuit(const char* fileName){
// Init all the variable to either nullptr or 0 for integers.
lastWireIndex = 0;
numberOfGates = 0;
numOfXorGates = 0;
numOfNotGates = 0;
numberOfInputParameters = 0;
numberOfOutputParameters = 0;
numberOfInputs = 0;
numberOfOutputs = 0;
gates.clear();
numOfInputsForEachParameter.clear();
numOfOutputsForEachParameter.clear();

// Read the file and fill the gates, number of parties, input indic
es, output indices and so on.
readCircuitFromFile(fileName);
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Ensure that the `readCircuitFromFile(fileName)` call is enclosed within a try-catch block to properly handle exceptions and log the error details.

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/2b48ceedf458c2c1797818c2fee1b4cfb2845964

and the following commits:

https://github.com/soda-mpc/soda-mpc/commit/c9b0f364bb62865b55ce1b7f2cfc5ee5cc7161dc

https://github.com/soda-mpc/soda-mpc/commit/8aef75240ffbe3d52b3d6953ffdf85e41bdd897a

https://github.com/soda-mpc/soda-mpc/commit/5b876ae3907957881206cfb8d8b715dcef0d5cc1

https://github.com/soda-mpc/soda-mpc/commit/08f9f311823c2a6f109f7b035ec9669b56001a06

F-2025-8807 - Memory Leak in garble() Function Due to Unfreed Allocations - Medium

Description:

File: *src/lib/crypto/circuit/circuitGarbler.cpp*

In the `CircuitGarbler::garble()` function, if the second allocation for `allOutputWireValues` fails, the pointer will be set to `nullptr`. However, the first allocation for `allInputWireValues` will have been successful. When a `bad_alloc` exception is thrown, the pointer to `allInputWireValues` is lost, leading to a memory leak since it is not freed.

```
tuple<block*, block*, vector<unsigned char> > CircuitGarbler::garbl
e(bool orderLabelsByPermutationBits, block *seed)
{
block *allInputWireValues = (block *)_aligned_malloc(sizeof(block)
* 2 * bc->getNumberOfInputs(), SIZE_OF_BLOCK);
if (allInputWireValues == nullptr) {
Logger::getInstance().critical("AllInputWireValues could not be all
ocated");
throw bad_alloc();
}
block *allOutputWireValues = (block *)_aligned_malloc(sizeof(block)
* 2 * bc->getNumberOfOutputs(), SIZE_OF_BLOCK);
if (allOutputWireValues == nullptr) {
Logger::getInstance().critical("AllOutputWireValues could not be al
located");
throw bad_alloc();
}
vector<unsigned char> permutationBits;
block seedLocl;
if (seed == nullptr) {
Commons::generateDeviceRandomBytes((unsigned char *)&seedLocl, SIZE
_OF_BLOCK);
seed = &seedLocl;
}

garble(allInputWireValues, allOutputWireValues, permutationBits, or
derLabelsByPermutationBits, *seed);
return make_tuple(allInputWireValues, allOutputWireValues, permutat
ionBits);
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation:

It is recommended to free the memory from the first allocation if the second allocation fails:

```
block *allInputWireValues = (block *)_aligned_malloc(sizeof(block)
* 2 * bc->getNumberOfInputs(), SIZE_OF_BLOCK);
if (allInputWireValues == nullptr) {
Logger::getInstance().critical("AllInputWireValues could not be all
ocated");
throw bad_alloc();
}
block *allOutputWireValues = (block *)_aligned_malloc(sizeof(block)
* 2 * bc->getNumberOfOutputs(), SIZE_OF_BLOCK);
if (allOutputWireValues == nullptr) {
Logger::getInstance().critical("AllOutputWireValues could not be al
located");
_aligned_free(allInputWireValues); // Free the first allocation bef
ore throwing
throw bad_alloc();
}
```

Resolution:

The issue is fixed by these 2 commits:

https://github.com/soda-mpc/soda-mpc/commit/86fcda96f944b8ecdd78dfcaab1bc0b6dda0cf01

https://github.com/soda-mpc/soda-mpc/commit/625f528e3b59e304327316456b40a9c5c56841cd

F-2025-8811 - Memory Leak if indexArray Allocation Fails - Medium

Description:

File: *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*

If `block* indexArray = (block *)_aligned_malloc(sizeof(block)* bc->getNumberOfOutputs(), SIZE_OF_BLOCK);` fails, there will be a memory leak due to the first allocation - `encryptedChunkKeys` not freed.

```
void FixedKeyCircuitEvaluator::verifyOutputWiresToNoFixedDelta(bloc
k *bothOutputsKeys, block* seed){
block* encryptedChunkKeys = (block *)_aligned_malloc(sizeof(block)*
bc->getNumberOfOutputs(), SIZE_OF_BLOCK);
if (encryptedChunkKeys == nullptr) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
throw bad_alloc();
}
block* indexArray = (block *)_aligned_malloc(sizeof(block)* bc->get
NumberOfOutputs(), SIZE_OF_BLOCK);
if (indexArray == nullptr) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc(); // Memory leak: encryptedChunkKeys not freed
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Severity: Medium

Recommendations

Remediation: Use RAII to handle the memory cleanup automatically.

```
void FixedKeyCircuitEvaluator::verifyOutputWiresToNoFixedDelta(bloc
k *bothOutputsKeys, block* seed){
// First allocation with RAII
unique_ptr<block, decltype(&_aligned_free)> encryptedChunkKeys(
(block *)_aligned_malloc(sizeof(block)* bc->getNumberOfOutputs(), S
IZE_OF_BLOCK),
_aligned_free
);
if (!encryptedChunkKeys) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
throw bad_alloc(); // encryptedChunkKeys automatically freed here
}
// Second allocation with RAII
unique_ptr<block, decltype(&_aligned_free)> indexArray(
(block *)_aligned_malloc(sizeof(block)* bc->getNumberOfOutputs(), S
IZE_OF_BLOCK),
_aligned_free
);
if (!indexArray) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc(); // Both encryptedChunkKeys and indexArray automa
tically freed here
}
```

Resolution:

The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/cbd4bf4e87250ed2b8b7655328b2d22ca938852f

F-2025-8813 - Memory management in garbleOutputWiresToNoFixedDelta - Medium

Description: File: *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*

A memory leak will occur if the second allocation for `indexArray` fails, as the previously allocated memory will not be freed.

```
void FixedKeyCircuitGarbler::garbleOutputWiresToNoFixedDelta(block
*deltaFreeXor, int nonXorIndex, block *emptyBothOutputKeys){
block* encryptedChunkKeys = (block *)_aligned_malloc(sizeof(block)*
numberOfOutputs, SIZE_OF_BLOCK);
if (encryptedChunkKeys == nullptr) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
throw bad_alloc();
}
block* indexArray = (block *)_aligned_malloc(sizeof(block)* numberO
fOutputs, SIZE_OF_BLOCK);
if (indexArray == nullptr) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc(); // Memory leak: encryptedChunkKeys not freed
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: It is recommended to allocate memory using RAII (Resource Acquisition Is Initialization) to ensure proper memory management and avoid memory leaks in case of allocation failures.

```
void FixedKeyCircuitGarbler::garbleOutputWiresToNoFixedDelta(block
*deltaFreeXor, int nonXorIndex, block *emptyBothOutputKeys){
if (!deltaFreeXor || !emptyBothOutputKeys) {
throw runtime_error("Null pointer arguments");
}
auto numberOfOutputs = bc->getNumberOfOutputs();
unique_ptr<block, decltype(&_aligned_free)> encryptedChunkKeys(
(block *)_aligned_malloc(sizeof(block)* numberOfOutputs, SIZE_OF_BL
OCK),
_aligned_free
);
if (!encryptedChunkKeys) {
Logger::getInstance().critical("EncryptedChunkKeys could not be all
ocated");
throw bad_alloc();
}
unique_ptr<block, decltype(&_aligned_free)> indexArray(
(block *)_aligned_malloc(sizeof(block)* numberOfOutputs, SIZE_OF_BL
OCK),
_aligned_free
);
if (!indexArray) {
Logger::getInstance().critical("IndexArray could not be allocated")
;
throw bad_alloc();
}
```

Resolution:

The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/df160285f002ef6bb644b9f5e51eb62785b125d7

F-2025-8814 - Potential Integer Overflow in getGarbledTableSize - Medium

Description: File: *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*

There is a potential integer overflow in the multiplication below:

```
int FixedKeyCircuitGarbler::getGarbledTableSize() {
if (isNonXorOutputsRequired == true) {
return sizeof(block) * ((bc->getNumberOfGates() - bc->getNumOfXorGa
tes() - bc->getNumOfNotGates()) * getNumOfRows() + 2 * bc->getNumbe
rOfOutputs());
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Use `size_t` and explicitly check for integer overflow before performing the multiplication.

```
size_t FixedKeyCircuitGarbler::getGarbledTableSize() {
size_t nonXorGates = bc->getNumberOfGates() - bc->getNumOfXorGates(
) - bc->getNumOfNotGates();
size_t numRows = getNumOfRows();

// Check for overflow in multiplications
if (nonXorGates > SIZE_MAX / numRows ||
nonXorGates * numRows > SIZE_MAX / sizeof(block)) {
throw runtime_error("Garbled table size overflow");
}
size_t baseSize = nonXorGates * numRows * sizeof(block);

if (isNonXorOutputsRequired) {
size_t outputs = bc->getNumberOfOutputs();
if (outputs > SIZE_MAX / (2 * sizeof(block)) ||
baseSize > SIZE_MAX - (2 * outputs * sizeof(block))) {
throw runtime_error("Garbled table size overflow with outputs");
}
return baseSize + (2 * outputs * sizeof(block));
}
return baseSize;
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/654506ce74a088a80c2c22ca3e9b3bc5a7a83222

F-2025-8817 - Absence of Input Validation - Medium

Description: File:

- *lib/src/crypto/circuit/garblingCircuit.cpp*
- *lib/src/crypto/circuit/circuitEvaluator.cpp*
- *lib/src/crypto/circuit/circuitGarbler.cpp*
- *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolEvaluator.cpp*
- *lib/src/protocols/twoEvaluatorsProtocol/twoEvaluatorsProtocolGarbler.cpp*
- *opcodesTranslator/opcodesToMPC.cpp*

There is missing input validation on `bc`. Null or invalid circuit pointers could lead to crashes or undefined behavior.

```
void GarblingCircuit::createCircuit(BooleanCircuit* bc, ...) {
this->bc = bc; // No null check
// ...
}
```

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Perform input validation to ensure that `bc` is not null and points to a valid circuit before use.

```
void GarblingCircuit::createCircuit(BooleanCircuit* bc, ...) {
if(!bc || bc->getNumberOfGates() == 0)
throw std::invalid_argument("Invalid circuit");
// ...
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/0f90794d0869e74b002ba329faae862fe9343504

F-2025-8825 - Potential Memory Leak in Constructor of HalfGatesCircuitEvaluator - Medium

Description: File: *lib/src/crypto/circuit/halfGatesEvaluator.cpp*

The constructor below creates a `BooleanCircuit` object but fails to delete it if createCircuitMemory throws an exception:

```
HalfGatesCircuitEvaluator::HalfGatesCircuitEvaluator(const char* fi
leName, bool isNonXorOutputsRequired) {
BooleanCircuit * bc = new BooleanCircuit(fileName); // Memory leak
if createCircuitMemory throws
HalfGatesCircuitEvaluator::createCircuitMemory(bc, isNonXorOutputsR
equired);
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Severity: Medium

Recommendations

Remediation: Replace the raw pointer with a smart pointer.

```
HalfGatesCircuitEvaluator::HalfGatesCircuitEvaluator(const char* fi
leName, bool isNonXorOutputsRequired) {
std::unique_ptr<BooleanCircuit> bc(new BooleanCircuit(fileName));
HalfGatesCircuitEvaluator::createCircuitMemory(bc.get(), isNonXorOu
tputsRequired);
bc.release();
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/df1f7d8468039d5727b717a405ab73cb2ca36dea

F-2025-8828 - Unsafe Array Access Leading to Potential Crash - Medium

Description:	File: *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp:101*

There is no bound check on the gate output index, which may lead to a buffer overflow:

```
computedWires[gates[i].output] = _mm_xor_si128(tempK0, tempK1);
```

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	2/5
Severity:	Medium

Recommendations

Remediation: Validate the index before accessing `computedWires`.

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/f3125be15ec80f338924614e5973d93859b98641

F-2025-8829 - Potential Memory Leak in Constructor of HalfGatesCircuitGarbler - Medium

Description:

File: *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*

A memory leak may occur if `createCircuitMemory` throws an exception. Additionally, it is advisable to validate `fileName` to ensure proper input, and it is recommended to use a smart pointer (e.g., `std::unique_ptr` or `std::shared_ptr`) instead of a raw pointer to manage memory more safely.

```
HalfGatesCircuitGarbler::HalfGatesCircuitGarbler(const char* fileNa
me, bool isNonXorOutputsRequired) {
BooleanCircuit* bc = new BooleanCircuit(fileName); // Raw pointer
createCircuitMemory(bc, isNonXorOutputsRequired);
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation:

A memory leak may occur if `createCircuitMemory` throws an exception. Additionally, it is advisable to validate `fileName` to ensure proper input, and it is recommended to use a smart pointer (e.g., `std::unique_ptr` or `std::shared_ptr`) instead of a raw pointer to manage memory more safely.

Resolution:

The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/df1f7d8468039d5727b717a405ab73cb2ca36dea

F-2025-8831 - Questionable Memory Cleanup on halfGatesCircuitGarbler.cpp - Medium

Description: File: *lib/src/crypto/circuit/halfGatesCircuitGarbler.cpp*

The memory cleanup mechanism in the code below may be considered insecure, depending on the security assumptions in place.

Sensitive cryptographic information is not properly wiped before being deallocated, potentially exposing this data in memory dumps. Additionally, the absence of an exception handler in the code introduces further concerns regarding robustness.

This pattern of memory deallocation is present across multiple files, and it is advisable to reassess whether a proper wiping of sensitive data should be implemented prior to deallocation.

This is a medium-severity item, as it could negatively affect the protocol by risking minor privacy breaches under specific conditions, but it's not catastrophic.

Impact: Failing to wipe sensitive data (e.g., `indexArray`, `encryptedChunkKeys`) before freeing leaves cryptographic secrets (e.g., wire labels, keys) in memory, potentially exposed via dumps or cold-boot attacks, compromising privacy; lack of exception handling risks leaks persisting.
Likelihood: Moderate—requires physical access or crash dump analysis post-execution; plausible in compromised environments but less so in good-faith setups.

```
HalfGatesCircuitGarbler::~HalfGatesCircuitGarbler(void) {
if (indexArray!=nullptr)
_aligned_free(indexArray);
if (encryptedChunkKeys!=nullptr)
_aligned_free(encryptedChunkKeys);
}
```

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	3/5
Severity:	Medium

Recommendations

Remediation: It is recommended to wipe the memory before freeing it to ensure that no sensitive data remains exposed in memory dumps. This practice helps mitigate the risk of cryptographic information being inadvertently captured or exposed.

```
HalfGatesCircuitGarbler::~HalfGatesCircuitGarbler(void) noexcept tr
y {
if (indexArray) {
volatile block* ptr = indexArray;
size_t size = bc->getNumberOfInputs();
for(size_t i = 0; i < size; i++) {
ptr[i] = _mm_setzero_si128();
}
_aligned_free(indexArray);
}
if (encryptedChunkKeys) {
volatile block* ptr = encryptedChunkKeys;
size_t size = bc->getNumberOfInputs();
for(size_t i = 0; i < size; i++) {
ptr[i] = _mm_setzero_si128();
}
_aligned_free(encryptedChunkKeys);
}
} catch (...) {
// Log error but don't throw from destructor
}
```

Resolution: The issue is fixed with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/5519ac07006f2d424b49b1c6bec2251f8887a613

https://github.com/soda-mpc/soda-mpc/commit/b49810c076c634b07cc0b7b2257e0ea1bbf4acea

https://github.com/soda-mpc/soda-mpc/commit/45316a60ca61910cf9956df0b5a574ff128879b4

https://github.com/soda-mpc/soda-mpc/commit/6520b20fdc0cb9c6309751362916dcee258df953

https://github.com/soda-mpc/soda-mpc/commit/67f9e371c0b174624a0b98d4946c8b52acc0cd53

https://github.com/soda-mpc/soda-mpc/commit/abadde973fc355c80b4b9fdca3b2834959ec3b9d

https://github.com/soda-mpc/soda-mpc/commit/f4a2669c2aaf3624fcfddd833fc0232a6819c844

https://github.com/soda-mpc/soda-mpc/commit/8df3a458379b61845442fdccdf0dd834fc04eb6f

https://github.com/soda-mpc/soda-mpc/commit/2a883908be352a382cb006cbc38d35e1ba7ee816

https://github.com/soda-mpc/soda-mpc/commit/fd614230c9c2574872d31650373d9538b70a4342 https://github.com/soda-mpc/soda-mpc/commit/8104757f68e85a4e244b730c483e8ce93ecb398d

https://github.com/soda-mpc/soda-mpc/commit/c1593a93158f67b29a94a1553a95320ab17d9291

https://github.com/soda-mpc/soda-mpc/commit/bb5e45c30774e814a6f148a9aebcc570090e742a

https://github.com/soda-mpc/soda-mpc/commit/8739cae576fa26f79861c28bb0fd02e406b37ef3

https://github.com/soda-mpc/soda-mpc/commit/e1e0bc9d71596d48c5c20f4b85bd9dc70a7ef115

https://github.com/soda-mpc/soda-mpc/commit/1d7cd38b610475023c1cb4eda3395474c459705f

F-2025-8940 - Sensitive Memory Data Not Properly Cleared in Destructor - Medium

Description: File: *lib/src/crypto/hash/hash.cpp*

The destructor frees the memory but does not wipe the sensitive information contained within it.

This is a medium-severity item, as it could negatively affect the protocol by risking minor privacy breaches under specific conditions, but it’s not catastrophic.

Impact: Failing to wipe sensitive data in `ctx` (e.g., hash state, keys) before freeing leaves cryptographic secrets in memory, potentially exposed via dumps or cold-boot attacks, compromising privacy. *Likelihood:* Moderate—requires physical access or crash dump post-execution; plausible in compromised setups but less so in good-faith environments.

```
OpenSSLHash::~OpenSSLHash(){
// Clean up
EVP_MD_CTX_free(ctx);
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Consider using `openssl_cleanse()` to wipe sensitive memory information.

A general observation: this codebase uses raw pointers, such as `(EVP_MD_CTX* ctx)` in this file, and manually calls `EVP_MD_CTX_free()` in the destructor. This approach may lead to memory leaks if exceptions occur before memory is cleaned up. Modern C++ recommends utilizing RAII for all resource management.

```
struct EVPMDCTXDeleter {
void operator()(EVP_MD_CTX* ctx) const {
```

```
if (ctx) {
EVP_MD_CTX_free(ctx);
}
}
};
using UniqueMDCTX = std::unique_ptr<EVP_MD_CTX, EVPMDCTXDeleter>;
// In OpenSSLHash:
UniqueMDCTX ctx;
```

Resolution: The issue is resolved with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/79faa3a7317d0ba0f151d5a6d2bcdaf0799564d7

https://github.com/soda-mpc/soda-mpc/commit/ce7693e67864f7a8aa2ef1457a56dda26de5d5e5

F-2025-9033 - Missing Channel Closure Mechanism in GarblerClient and GCClient - Medium

Description:

Files:

- *opcodesTranslator/garblerClient.hpp*
- *opcodesTranslator/gcClient.hpp*

In the client classes (`GarblerClient`, `GCClient`), gRPC channels are instantiated, but there is no mechanism to properly shut them down. Without explicit cleanup, resources may not be released efficiently, potentially leading to resource leaks or undefined behavior.

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	3/5
Severity:	Medium

Recommendations

Remediation:

To address the issue, ensure that proper resource cleanup is implemented in the destructors of both the `GarblerClient` and `GCClient` classes. This will ensure that gRPC channels are gracefully shut down and any resources are properly released when the objects are destroyed.

Resolution:

The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/5be69f169b0f132092f25de96374829dc4939bfa

F-2025-9197 - Hardcoded FixedKey Violates Best Practices - Medium

Description:

Files:

- *lib/include/crypto/circuit/fixedKeyCircuitGarbler.hpp*
- *lib/src/crypto/circuit/fixedKeyCircuitGarbler.cpp*

In line 14 of the header file and line 12 of the source file, a fixed AES key is hardcoded into the code:

```
block fixedKey; // Fixed-key AES used for the garbling process. Thi
s will be hardcoded unless the user changes it. The fixed key
// optimization reduces the number of the costly setKey function of
the AES prp and thus optimize the performance.
AES_KEY aesFixedKey;
```

```
// Set the fixed key.
fixedKey = _mm_set_epi8(36, -100, 50, -22, 92, -26, 49, 9, -82 , -8
6, -51, -96, 98, -20, 29, -13);
```

While the motivation for optimizing performance by using a fixed AES key is understandable, embedding a cryptographic key directly in the source code represents a significant security risk. In the context of the Garbled Circuit protocol, knowledge of the AES key would allow an attacker to potentially decrypt garbled tables or precompute values, undermining the privacy and integrity of the protocol. Furthermore, the code containing the key may be accessible across the company, making it difficult to trace the source of any potential leaks.

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Severity: Medium

Recommendations

Remediation: Treat `fixedKey` as a secret parameter:

- Generate it randomly at runtime by the garbler (recommended for simplicity and enhanced security).

- Provide documentation on how the `fixedKey` is generated.

Revised Approach:

- Remove the hardcoded key value from the source code.
- Initialize `fixedKey` with a secure random value at runtime.
- Set `aesFixedKey` once and reuse it to optimize performance without compromising security.

Resolution: According to the client:

A fixed is incorporated. The fixed does two things:
1. Replaces the fixed-key per circuit - thereby drastically minimizes the computational advantage of an adversary (which is measured by the number of times the same fixed-key is used) (See paper by Jonathan Katz et al.)
2. Move the seed from which the fixed-keys are derived from being a hardcoded to a configurable parameter at the launch of the node (all nodes - garblers and evaluators - have the same seed. Again, this does not incurs a secrecy risk as per the TCCR assumption of AES).

F-2025-9219 - Memory Leak in Constructor due to Unreleased Resources - Medium

Description:

File: *lib/src/protocols/twoEvaluatorsProtocolGarbler.cpp:18-19*
The issue is that `bc` is allocated but not deleted. `init` passes it to `HalfGatesCircuitGarbler`, which stores it in `this->bc`, but `~GarblingCircuit` does not free it, resulting in a memory leak.

This is a medium-severity item, as it could negatively affect the protocol by causing memory leaks under specific conditions, but it’s not catastrophic.

Impact: Failing to delete bc in `~HalfGatesCircuitGarbler` leaks memory each time a `TwoEvaluatorsProtocolGarbler` is created and destroyed, potentially degrading node performance over time (e.g., memory exhaustion), disrupting garbling if unchecked.
Likelihood: High—memory leaks occur with every instance destruction; common in long-running systems with frequent garbler instantiations.

```
TwoEvaluatorsProtocolGarbler::TwoEvaluatorsProtocolGarbler(string c
ircuitName, int N, int T, unsigned char* evaluator0ItMacKey, unsign
ed char* evaluator1ItMacKey) {
BooleanCircuit* bc = new BooleanCircuit(circuitName.c_str());
init(bc, N, T, evaluator0ItMacKey, evaluator1ItMacKey, evaluator0_s
eed, evaluator1_seed, sharesSeed);
}
```

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 4/5

Severity: Medium

Recommendations

Remediation: Ensure that `bc` is properly owned and freed by the appropriate component.

```
TwoEvaluatorsProtocolGarbler::~TwoEvaluatorsProtocolGarbler() {
// Implicit via unique_ptr<ProtocolGarblerData>, but ensure bc is f
reed
}
void TwoEvaluatorsProtocolGarbler::init(BooleanCircuit* bc, /* ...
*/) {
// ... existing code ...
garblerDataPtr->garbler = make_unique<HalfGatesCircuitGarbler>(bc);
// bc owned by garbler
// No delete bc here
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/597886e77607864d4ad13d8ee5169311a2f405f0

F-2025-8689 - Potential Race Condition Between Directory Existence Check and Creation - Low

Description: File: *lib/src/commons/logger.cpp*

A potential race condition exists between the call to `filesystem::exists("logs")` and `filesystem::create_directory("logs")`. During this interval, another thread or process may create or delete the `logs` directory, which could lead to unexpected behavior or failure.

```
void Logger::setName(string name, string pathToConfigFile) {
try {
filesystem::create_directories("logs"); // Creates if doesn't exist
, no race condition
} catch (const filesystem::filesystem_error& e) {
throw runtime_error("Failed to create logs directory: " + string(e.
what()));
}
// ... rest of code ...
}
```

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 4/5

Severity: Low

Recommendations

Remediation: To eliminate the race condition, it is advisable to use the `filesystem::create_directory` function directly, without pre-checking the directory's existence. This function is designed to handle the race condition gracefully:

- If the directory already exists, `create_directory` will return `false`.
- If the directory does not exist, it will be created and `create_directory` will return `true`.
- If the directory cannot be created (e.g., due to permission issues), it will either throw an exception or return `false`.

This approach avoids the potential inconsistency between checking for existence and creating the directory, thus mitigating the race condition.

```
void Logger::setName(string name, string pathToConfigFile) {
try {
filesystem::create_directories("logs"); // Creates if doesn't exist
, no race condition
}
catch (const filesystem::filesystem_error& e) {
throw runtime_error("Failed to create logs directory: " + string(e.
what()));
}
// ... rest of code ...
}
```

Resolution: The following 2 commits resolved the issue:

https://github.com/soda-mpc/soda-mpc/commit/497403712f2fb5f8e0f3c77947c502967c72cd1a

https://github.com/soda-mpc/soda-mpc/commit/fa783a13af57b6ee6e2f56fe22b3ee33288aafb4

F-2025-8694 - Unsafe File Operation in readCircuitFromFile - Low

Description: File: *lib/src/crypto/circuit/booleanCircuit.cpp*

The function `readCircuitFromFile` accepts a `fileName` parameter without performing validation or sanitization. This allows an attacker to provide a malicious file name, potentially gaining access to sensitive files outside the intended directory, such as supplying a path like `../../etc/passwd`.

```
void BooleanCircuit::readCircuitFromFile(const char* fileName)
{
ifstream myfile;
myfile.open(fileName);
if (!myfile.is_open()) {
Logger::getInstance().critical("Circuit file " + string(fileName) +
"not found");
throw runtime_error("Circuit file " + string(fileName) + "not found
");
}
//…
}
```

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: It is recommended to implement

- file path sanitization
- file name sanitization to ensure the input contains only valid alphanumeric characters or specific file types (e.g., circuit files).

This approach will help prevent potential security vulnerabilities, such as directory traversal attacks.

Resolution: The issue is resolved with the following commits:

https://github.com/soda-mpc/soda-mpc/commit/b39e3d92f688e408c3d83165608acf1d96611c0d https://github.com/soda-mpc/soda-mpc/commit/39d37c65e2eef6915aaf0b39ba386e8ef4efaf00

https://github.com/soda-mpc/soda-mpc/commit/d48f11d0d959150c0e3e7a096e0d1ef9b00e6f25

F-2025-8808 - Side-Channel Information Leak - Low

Description: File: *src/lib/crypto/circuit/circuitGarbler.cpp:58-62*

The random seed is only generated when the `seed` is `nullptr`, which introduces branching in the code. This approach does not guarantee constant-time behavior, as timing differences could potentially reveal information about the value of `seed`.

```
block seedLocl;
if (seed == nullptr) {
Commons::generateDeviceRandomBytes((unsigned char *)&seedLocl, SIZE
_OF_BLOCK);
seed = &seedLocl;
}
```

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: Use constant-time behavior to avoid timing attacks in cryptographic code, such as below:

```
// Always generate a new seed and use constant-time selection
block seedLocal;
Commons::generateDeviceRandomBytes((unsigned char *)&seedLocal, SIZ
E_OF_BLOCK);
seed = (seed != nullptr) ? seed : &seedLocal; // Constant-time sele
ction
```

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/e17baf9b078f346e4642d1147931a6634da23f97

F-2025-8809 - Side-Channel Information Leak in Signal Bit Computation - Low

Description: File: *src/lib/crypto/circuit/fixedKeyCircuitEvaluator.cpp:73-81*

In the `FixedKeyCircuitEvaluator::compute()` function, the `getSignalBitOf()` operation is non-constant time, which could lead to potential timing leaks, exposing sensitive information about the signal bits.

```
// Get the signal bits of A and B which are the input keys computed
.
int a = getSignalBitOf(A);
int b = getSignalBitOf(B);
// Calc the tweak
block tweak = _mm_set_epi32(0,0,0,i);
// Deduce the key to encrypt
block key = _mm_xor_si128(_mm_xor_si128(twoA, fourB), tweak);
```

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Low

Recommendations

Remediation: Create constant time function to avoid the side channel leak.

```
// Use constant-time operations for signal bit extraction and key c
omputation
unsigned char getSignalBitConstantTime(const block& input) {
unsigned char lsb;
_mm_store_ss((float*)&lsb, _mm_castsi128_ps(input));
return lsb & 1; // Constant time AND
}
// In compute():
unsigned char a = getSignalBitConstantTime(A);
unsigned char b = getSignalBitConstantTime(B);
block tweak = _mm_set_epi32(0,0,0,i);
block key = _mm_xor_si128(_mm_xor_si128(twoA, fourB), tweak);
```

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/566a8edf39408e98231d7ecd844daed1e752e3e1

F-2025-8812 - Potential Integer Overflow in getGarbledTableSize - Low

Description: File: *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*

`getGarbledTableSize()` lacks safeguards against integer overflow, as its calculations involve multiplying several potentially large integers. If any intermediate result exceeds the maximum value representable by a 32-bit `int` (2^{31} - 1, approximately 2.1 billion), an overflow will occur, causing unintended wraparounds and incorrect results.

```
int FixedKeyCircuitEvaluator::getGarbledTableSize()
{
if (isNonXorOutputsRequired == true) {
return sizeof(block) * ((bc->getNumberOfGates() - bc->getNumOfXorGa
tes() - bc->getNumOfNotGates()) * getNumOfRows() + 2 * bc->getNumbe
rOfOutputs());
}
else {
return sizeof(block) * (bc->getNumberOfGates() - bc->getNumOfXorGat
es() - bc->getNumOfNotGates()) * getNumOfRows();
}
}
```

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: It is recommended to use `size_t` instead of `int`, as it provides a 64-bit integer on most platforms instead of a 32-bit one. Additionally, explicitly guard against integer overflow to ensure robustness.

```
size_t FixedKeyCircuitEvaluator::getGarbledTableSize() {
size_t nonXorGates = bc->getNumberOfGates() - bc->getNumOfXorGates(
) - bc->getNumOfNotGates();
size_t numRows = getNumOfRows();

// Check for overflow
if (nonXorGates > SIZE_MAX / numRows ||
nonXorGates * numRows > SIZE_MAX / sizeof(block)) {
throw runtime_error("Garbled table size overflow");
}
size_t baseSize = nonXorGates * numRows * sizeof(block);

if (isNonXorOutputsRequired) {
size_t outputs = bc->getNumberOfOutputs();
if (outputs > SIZE_MAX / (2 * sizeof(block)) ||
baseSize > SIZE_MAX - (2 * outputs * sizeof(block))) {
throw runtime_error("Garbled table size overflow with outputs");
}
return baseSize + (2 * outputs * sizeof(block));
}

return baseSize;
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/654506ce74a088a80c2c22ca3e9b3bc5a7a83222

F-2025-8824 - Lack of Error Handling in AES Encryption Functions - Low

Description: Files:

- *lib/src/crypto/circuit/fixedKeyCircuitEvaluator.cpp*
- *lib/src/crypto/circuit/FixedKeyCircuitGarbler.cpp*
- *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*
- *lib/src/commons/commons.cpp*

Across all the files mentioned above, which implement AES operations, there is an absence of error handling. This results in silent failures in scenarios such as incorrect key setup or hardware malfunctions. When such failures occur, the code proceeds with invalid encrypted values, potentially compromising the application's integrity and security.

```
void FixedKeyCircuitEvaluator::computeOutputWiresToNoFixedDelta(...
) {
// ...
AES_encryptC(&key, &encryptedKey, &aesFixedKey); // No error check
// ...
}
```

```
void FixedKeyCircuitGarbler::garbleOutputWiresToNoFixedDelta(...) {
// ...
AES_ecb_encrypt_chunk_in_out(indexArray, encryptedChunkKeys,
numberOfOutputs, &aesSeedKey); // No error check
// ...
}
```

Status: Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	1/5
Severity:	Low

Recommendations

Remediation: It is recommended to:

- **Check the result of encryption**: Always verify the success of encryption operations to ensure that no errors occurred during the process.

- **Log errors**: In the event of a failure (e.g., bad key setup, hardware failure), log detailed error information to facilitate troubleshooting and improve visibility into the failure.
- **Halt operations**: Instead of continuing with potentially invalid or compromised encrypted data, halt further operations when an error is detected to prevent the propagation of faulty or insecure data through the system.

```
// For AES_encryptC
int rc = AES_encryptC(&key, &encryptedKey, &aesFixedKey);
if(rc != 0) {
Logger::getInstance().securityAlert("AES encryption failure");
throw CryptoOperationFailed("AES encryption error: " + std::to_stri
ng(rc));
}
// For AES_ecb_encrypt_chunk_in_out
if(AES_ecb_encrypt_chunk_in_out(...) != SUCCESS_CODE) {
Logger::getInstance().securityAlert("AES ECB encryption failure");
throw CryptoOperationFailed("AES ECB chunk encryption failed");
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/ead010bfed1ba94e33ec4b86fcd1d748849bf5fe

F-2025-8883 - Missing Input Validation in OpenSSL_ECDSA::sign() - Low

Description: File: *lib/src/crypto/digitalSignature/digitalSignature.cpp*

The code below lacks the input validation on the message.

```
void OpenSSL_ECDSA::sign(vector<unsigned char> & message, vector<un
signed char> & signature) {
if (!canSign) {
Logger::getInstance().critical("Error: ECDSA cannot be used to sign
, no signing key found");
throw runtime_error("Error: ECDSA cannot be used to sign, no signin
g key found");
}
if (1 != EVP_DigestSignInit(ctx, nullptr, EVP_sha256(), nullptr, pk
ey)) {
Logger::getInstance().critical("Error initializing digest sign");
throw runtime_error("Error initializing digest sign");
}
// Ensure that the signature vector has enough space
if (signature.size() < MAX_SIGNATURE_SIZE) {
signature.resize(MAX_SIGNATURE_SIZE);
}
size_t sigSize = signature.size();
// Sign the message using openssl implementation.
// sigSize should be set to the size of the signature array and is
changed during the call to the actual signature size
if (EVP_DigestSign(ctx, signature.data(), &sigSize, message.data(),
message.size()) != 1) {
Logger::getInstance().critical("Error creating signature");
throw runtime_error("Error creating signature");
}
// Resize the signature vector to the actual signature size
signature.resize(sigSize);
}
```

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: Ensure the message is not empty to prevent signing an empty message:

```
void OpenSSL_ECDSA::sign(vector<unsigned char> & message, vector<un
signed char> & signature) {
if (!canSign) {
throw runtime_error("No private key available for signing");
}
if (message.empty()) {
    throw invalid_argument("Empty message");
  }
  // …
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/70e0d28ef4c5d1ffdab2ba4939d5d01ea5074ab8

F-2025-8928 - Absence of Secure Memory Data Wiping - Low

Description: File: *lib/src/crypto/hash/hash.cpp*

The `EVP_MD_CTX` structure may contain sensitive data, such as intermediate hash states or keys for HMAC. While `EVP_MD_CTX_free()` frees the memory allocated for the structure, it does not securely wipe the sensitive data contained within it.

```
OpenSSLHash::~OpenSSLHash(){
// Clean up
EVP_MD_CTX_free(ctx);
}
```

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Low

Recommendations

Remediation: Implement a more robust destructor that securely wipes sensitive data before deallocating memory.

```
~OpenSSLHash() {
if (ctx != nullptr) {
EVP_MD_CTX_reset(ctx); // Clears internal state
EVP_MD_CTX_free(ctx);
}
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/79faa3a7317d0ba0f151d5a6d2bcdaf0799564d7

F-2025-8826 - Missing Input Validation Leading to Potential Buffer Overflow - Info

Description: File: *lib/src/crypto/circuit/halfGatesCircuitEvaluator.cpp*

The code below lacks validation of input pointers and proper size checks, which can lead to buffer overflow vulnerabilities.

```
void HalfGatesCircuitEvaluator::compute(block * singleWiresInputKey
s, block * output) {
for (int i = 0; i < numberOfInputs; i++){
computedWires[i] = singleWiresInputKeys[i]; // No null check or bou
nds validation
}
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Add input validation to the code logic.

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/76fdf89bce7deee36bb74534fa48717da157c397

F-2025-8838 - Missing Documentation on Security Assumptions in TedKrovetzAesNiWrapperC.cpp - Info

Description:	The file *lib/src/crypto/circuit/TedKrovetzAesNiWrapperC.cpp* lacks documentation both at the file level and for individual functions/methods.
Status:	Fixed

Classification

Severity:	Info

Recommendations

Remediation:

Add documentation outlining the security assumptions for this file.

File-level Documentation:

- Clearly specify whether this file is intended to be a low-level library.
- Highlight that the implementation is performance-focused and intended for internal use.
- Emphasize that input validation is the responsibility of the caller.
- Document any assumptions regarding memory alignment, buffer sizes, and related constraints.

Function-level documentation:

```
/**
* Low-level AES encryption implementation using AES-NI instructions
.
*
* SECURITY REQUIREMENTS:
* - Caller MUST validate all input parameters
* - Input/output blocks MUST be 16-byte aligned
* - Key material MUST be properly sanitized by caller
* - No protection against fault attacks - caller must implement if
needed
*
* @param blks Pointer to aligned input blocks
* @param aesKey Pointer to initialized AES key structure
*/
void AES_ecb_encrypt_blks_4(block *blks, AES_KEY *aesKey);
```

Resolution:

The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/1964e4777964790a3243807f2bbab9922be04a8d

F-2025-8887 - DER Format Validation Issues - Info

Description:

File: *lib/src/crypto/encryption/cryptoppRsa.cpp*

The `isDERFormat()` method currently logs a critical error but returns `false`. This approach may lead callers to overlook the return value, resulting in inconsistent error handling across the codebase - some instances throw exceptions, while others return `false`.

```
bool CryptoPP_RSAOAEP::isDERFormat(const vector<unsigned char>& der
Bytes) {
try {
CryptoPP::RSA::PublicKey publicKey;
CryptoPP::StringSource ss(...);
publicKey.BERDecode(ss);
return true;
} catch (const CryptoPP::Exception& e) {
Logger::getInstance().critical("Error: " + string(e.what()));
return false;
}
}
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Make error handling consistent, use `debug()` instead of `critical()`.

```
bool CryptoPP_RSAOAEP::isDERFormat(const vector<unsigned char>& der
Bytes) {
try {
CryptoPP::RSA::PublicKey publicKey;
CryptoPP::StringSource ss(...);
publicKey.BERDecode(ss);
return true;
} catch (const CryptoPP::Exception& e) {
Logger::getInstance()..debug("DER format validation failed: " + str
ing(e.what()));
return false;
}
}
```

Resolution: The issue is solved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/144580ecd132991cdff5e78ee12c28cc5e85af08

F-2025-8937 - Unsafe Casting Using const_cast on String Literals - Info

Description: File: *lib/src/crypto/hmac/hmac.cpp:36*

While the code may function correctly today, using `const_cast` to remove the `const` qualifier from a pointer (i.e., converting `const char*` to `char*`) bypasses the compiler's safety checks, which is not recommended. This could lead to unintended side effects, and the function does not appear to modify the string.

```
OSSL_PARAM_construct_utf8_string("digest", const_cast<char*>("SHA25
6"), 0),
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Instead of using `const_cast`, consider making the string mutable before passing it in. This will maintain the integrity of the original data and avoid bypassing compiler safety checks.

```
char digestName[] = "SHA256";
OSSL_PARAM params[] = {
OSSL_PARAM_construct_utf8_string("digest", digestName, 0),
OSSL_PARAM_END
};
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/0e228e31791a7aa04b1ee79550efdf5a54985f45

F-2025-8986 - Insecure Channel - Info

Description:

Files:

- *opcodesTranslator/kms/kmsClient.cpp*
- *opcodesTranslator/kms/kmsServer.cpp*

KMSServer::runServer, line 23:

`builder.AddListeningPort(ServerAddressStr, grpc::InsecureServerCredentials());`

KMSClient::KMSClient, lines 15 - 21:

`grpc::InsecureChannelCredentials()` creates a channel that is both unencrypted and unauthenticated. As a result, data transmitted over this channel is susceptible to eavesdropping and tampering attacks.

Additionally, this approach does not authenticate the server to which the client is connecting, potentially exposing the client to a connection with a malicious server.

Even in scenarios where the client and server are hosted on the same machine, it is still recommended to implement encryption and authentication mechanisms to ensure data confidentiality and integrity.

```
// Create a gRPC channel
auto channel = grpc::CreateChannel(
serverAddressStr,
grpc::InsecureChannelCredentials()
);

// Create a stub (client)
stub_ = KMSService::NewStub(channel);
```

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: To enhance security, it is recommended to use secure authentication mechanism for both the server and client channels, ensuring encrypted communication and server authentication.

Resolution: While we recommend servers and clients authenticate each other and communicate entirely via encrypted channels, the client is comfortable with encrypting the critical information and authenticating the server with a digital signature.

F-2025-9178 - Non-Idiomatic Exception Handling in gRPC Service Method - Info

Description: File: *lib/src/communication/streamComm:228*

Throwing an exception in `BiDirectionalStreamComm::CommunicationServiceImpl::Communicate` is inconsistent with gRPC's design, which expects errors to be handled via `Status` returns rather than exceptions. Refer to the gRPC error handling guide for best practices.

```
if (stream->Read(&message)) {
Logger::getInstance().info("Server received init message");
} else {
Logger::getInstance().error("Failed to read init message from the c
lient");
throw CommunicationException("Failed to read init message from the
client");
}
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Adopt gRPC's standard error-handling approach by returning an appropriate `Status` code instead of throwing an exception.

```
Status BiDirectionalStreamComm::CommunicationServiceImpl::Communica
te(ServerContext* context, ServerReaderWriter<CommunicationMessage,
CommunicationMessage>* stream) {
serverContext = context;
serverStream = stream;
CommunicationMessage message;
if (!stream->Read(&message)) {
Logger::getInstance().error("Failed to read init message from the c
lient");
return Status(StatusCode::UNAVAILABLE, "Failed to read init message
");
}
Logger::getInstance().info("Server received init message");
// Rest of the function...
}
```

Resolution: The issue is fixed with this commit:

https://github.com/soda-mpc/soda-mpc/commit/d553897da71c15db36686e706613f5bcad6b49c4

F-2025-9193 - Unimplemented getInputWireIndices Function - Info

Description: File: *lib/include/crypto/circuit/booleanCircuit.hpp:75*

The function below is declared but lacks an implementation in the provided `.cpp` file. While it is not currently in use, its absence may lead to linkage errors if referenced in the future.

```
vector<int> getInputWireIndices(int parameterNumber);
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Either remove the declaration if the function is unnecessary or provide an implementation to prevent potential linkage errors.

Resolution: The issue is resolved with this commit:

https://github.com/soda-mpc/soda-mpc/commit/2a90f254a00bae40b9c0d8dead4e372fbe554715

F-2025-9216 - Incomplete Verification in internalVerify - Info

Description:

File: lib/src/crypto/circuit/halfGatesEvaluator.cpp

line 235 - 237

```
for (int index0 = 0; index0 < 2; index0++) {
for (int j = 0; j < 2; j++) {
if (index0 1 && j 1) continue; // Skips 1,1 case
// ... compute k0 ...
}
}
```

Issue: skips verification of the (1,1) input case for AND gates, assuming only the 0-wire (k0) needs checking. In Half Gates, internalVerify must confirm all four input combinations (00, 01, 10, 11) produce consistent outputs per the garbled table, not just k0. This weakens verification.

In Half Gates (Zahur et al., 2015):

Garbling: An AND gate uses two garbled table rows:
Row 1: Adjusts the output based on input A's signal bit.
Row 2: Adjusts based on input B's signal bit.
Evaluation: For inputs (a, b), the evaluator computes:
tempK0 = a ? (H(A) XOR garbledTables[0]) : H(A) (garbler's half).
tempK1 = b ? (H(B) XOR garbledTables[1] XOR A) : H(B) (evaluator's half).
Output = tempK0 XOR tempK1.
Verification: The garbler provides both input keys (e.g., A0, A1 = A0 XOR delta, B0, B1 = B0 XOR delta) and tables. internalVerify must:
Compute outputs for all four combinations (00, 01, 10, 11).
Ensure consistency with the garbled table and Free XOR (e.g., deltaFreeXor).

Truth Table for AND:

00 → 0 (k0)
01 → 0 (k0)
10 → 0 (k0)
11 → 1 (k1)

The current code logic:

Computes k0 for (0,0), (0,1), (1,0), verifies they match.
Skips (1,1), assuming k1 isn't needed since it's the only 1-output and AND-specific.

(0,0): Sets garbledWires[gates[i].output] = k0.
(0,1), (1,0): Computes k0 again, compares to stored value.

(1,1): Skipped via if (index0 1 && j 1) continue;.

Problem: Skipping (1,1) assumes the table correctly produces k1, but a malformed table could pass verification if only k0 cases are checked, weakening security.

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: The code comment is correct—there's only one k1 instance (11), so without an external reference (e.g., expected output keys or translation table), direct comparison is tricky.

Recommendation: use the garbled output wire's signal bit and deltaFreeXor to infer k1's correctness:
If k0 is the 0-wire key, k1 = k0 XOR deltaFreeXor must match the computed (1,1) output if Free XOR holds.

```
block k0, k1;
for (int index0 = 0; index0 < 2; index0++) {
wire0SignalBit = getSignalBitOf(inputs[index0]);
for (int j = 0; j < 2; j++) {
wire1SignalBit = getSignalBitOf(inputs[j + 2]);
// ... compute tempK0, tempK1 (same as original) ...
block output = _mm_xor_si128(tempK0, tempK1);
if (index0 == 0 && j == 0) {
k0 = output;
garbledWires[gates[i].output] = k0;
} else if (index0 == 1 && j == 1) {
k1 = output;
// Check if k1 = k0 XOR deltaFreeXor (Free XOR consistency)
if (!equalBlocks(k1, _mm_xor_si128(k0, deltaFreeXor))) {
return false;
}
} else if (!equalBlocks(output, k0)) {
return false;
}
}
}
```

Resolution: While `internalVerify` isn't used at the current stage, the client would take our input to implement rigorous and complete circuit verification when the time comes.

F-2025-9217 - Insufficient Documentation - Info

Description: File: *lib/include/protocols/twoEvaluatorsProtocolGarbler.hpp:7-10*

The documentation lacks detailed explanations on how IT-MACs and shares provide security against collusion. This absence increases the risk of misuse, such as the potential use of weak key_s._

```
/**
* TODO add documentation
*/
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Document the mechanisms by which IT-MACs and shares safeguard against collusion to mitigate the risk of misuse and weak keys.

Resolution: The issue is resolved with this fix:

https://github.com/soda-mpc/soda-mpc/commit/ea6604e88dbdb5ae07336ff856498f5ecb15e2e8

Observation Details

Disclaimers

Hacken Disclaimer

The blockchain protocol given for audit has been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in the protocol source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other protocol statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the blockchain protocol.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Blockchain protocols are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the protocol can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited blockchain protocol.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/soda-mpc/soda-mpc
Commit	7fb3a27404df5720bf557eca2f0e19cd56081458

The remediation check has been conducted based on commit hash 649eb99, which reflects the status of each issue following this process. It is important to acknowledge that this commit may include changes made subsequent to the initial review commit, which were not part of the audit assessment.