

# **Report: Privacy-Preserving Multi-Cloud GRC Integration Using Homomorphic Encryption and Secure Multi-Party Computation for Threat Intelligence Sharing**

## **1. Executive Summary**

This report summarizes the implementation of a regulatory-compliant, homomorphic encryption-based GRC simulation developed by Yashwardhan Singh. The simulation leverages Python and the TenSEAL library to prototype secure threat intelligence aggregation from multiple cloud providers while enforcing jurisdictional boundaries. Cloud-specific encrypted scores for threat levels (Low, Medium, High) are shared with a central GRC server. Data from unauthorized jurisdictions (e.g., "IN") is rejected to simulate GDPR-like enforcement. The system logs compliance status and final decrypted results after aggregation, ensuring privacy preservation and compliance validation. All modules—including CloudProvider, CentralGRCServer, Decryptor, and encryption context handlers—were integrated and tested in the simulation, producing encrypted data flow logs and final decrypted reports with performance timings.

## **2. Objective and Scope**

The objective of this task is to implement a privacy-preserving multi-cloud GRC simulation using homomorphic encryption to enable secure threat intelligence sharing while enforcing regulatory jurisdiction boundaries.

The scope includes:

- Developing a prototype GRC framework where multiple cloud providers encrypt threat scores using a public encryption context.
- Simulating a central GRC server that enforces compliance by accepting encrypted data only from allowed jurisdictions.
- Aggregating encrypted threat data from authorized clouds and decrypting the aggregated result securely.
- Logging compliance decisions and final decrypted threat reports along with performance metrics.

### 3. Technical Survey

#### 3.1 Homomorphic Encryption (HE)

Homomorphic Encryption (HE) is a cryptographic technique that allows computations to be performed directly on encrypted data without requiring decryption. This property preserves data privacy throughout the processing lifecycle, making HE especially suitable for secure data analytics and compliance tasks in untrusted environments such as public clouds.

There are several types of HE schemes:

- **Partially Homomorphic Encryption (PHE):** Supports either addition or multiplication on ciphertexts (e.g., RSA, ElGamal).
- **Somewhat Homomorphic Encryption (SHE):** Allows limited operations of both addition and multiplication.
- **Fully Homomorphic Encryption (FHE):** Supports arbitrary computations on encrypted data, though often with high computational overhead.

Notable libraries implementing HE include Microsoft SEAL, IBM HELib, and PALISADE. For GRC use cases, schemes like BFV and CKKS are popular for encrypted numeric computations.

#### 3.2 Secure Multi-Party Computation (SMPC)

Secure Multi-Party Computation enables multiple parties to jointly compute a function over their inputs while keeping those inputs private. SMPC protocols ensure that no party learns anything beyond the intended output, which is valuable for collaborative threat intelligence sharing across different cloud providers or organizations.

Common SMPC approaches include:

- **Yao's Garbled Circuits:** Efficient for two-party computations.
- **Secret Sharing-based Protocols (e.g., SPDZ):** Suitable for multi-party setups with stronger security guarantees.

Despite their strong privacy properties, SMPC protocols often face challenges related to communication overhead and scalability in complex cloud environments.

#### 3.3 Applicability to Multi-Cloud GRC

Both HE and SMPC provide mechanisms to address the privacy challenges of cross-cloud GRC operations, such as encrypted data aggregation, real-time compliance checks, and secure threat intelligence sharing. This survey informed the decision to focus on Homomorphic Encryption for

the prototype framework due to its relative maturity, ease of integration, and better performance characteristics for the envisioned encrypted analytics workflows.

## 4. Proposed GRC Framework

### 4.1 Overview

This section presents a conceptual framework for integrating Homomorphic Encryption (HE) into multi-cloud Governance, Risk, and Compliance (GRC) operations. The framework is designed to enable privacy-preserving threat intelligence sharing and compliance reporting across heterogeneous cloud environments without exposing raw or sensitive data.

### 4.2 Architecture Components

- **Encrypted Data Sources:** Various cloud platforms and organizational endpoints generate encrypted telemetry and threat intelligence data using HE schemes. Data remains encrypted before transmission.
- **Homomorphic Computation Engine:** A centralized or distributed computation module performs analytics directly on encrypted data, leveraging HE's ability to compute without decryption.
- **Compliance Monitor:** This component assesses encrypted analytic results against regulatory thresholds and policies (e.g., GDPR, CCPA) to ensure ongoing compliance without raw data exposure.
- **Key Management Service (KMS):** Responsible for secure generation, distribution, and rotation of encryption keys, ensuring proper access controls and minimizing trust assumptions.
- **Audit and Logging Module:** Maintains an immutable record of encrypted data transactions and compliance actions for forensic and regulatory purposes.

### 4.3 Data Flow

- Data owners encrypt sensitive GRC data locally using HE and send it to the computation engine.
- The computation engine performs analytics and aggregation on ciphertexts, generating encrypted results.

- The compliance monitor evaluates these results to flag any non-compliance or risk conditions.
- Alerts and compliance reports are generated without decrypting the underlying data.
- Authorized parties may decrypt results with appropriate keys for further investigation if necessary.

#### **4.4 Privacy and Security Considerations**

- Data confidentiality is preserved end-to-end, reducing risk from insider threats or compromised cloud providers.
- Key management is critical; the framework assumes a robust KMS with strict access policies.
- The design supports scalability across multiple cloud vendors and regulatory jurisdictions.

### **5. Implementation Details**

The proposed Privacy-Preserving Multi-Cloud GRC framework is implemented in Python using the TenSEAL library, leveraging homomorphic encryption (HE) to enable secure threat intelligence sharing across multiple cloud providers.

#### **5.1 CentralGRCServer Module**

The CentralGRCServer class is responsible for receiving encrypted threat intelligence data from multiple cloud providers and performing encrypted aggregation without decrypting the data. It maintains a list of encrypted vectors and adds them homomorphically.

```
class CentralGRCServer:
```

```
    def __init__(self):
```

```
        self.encrypted_batches = []
```

```
    def receive_score(self, provider_name, jurisdiction, encrypted_vector):
```

```
        print(f'[GRC] Received encrypted data from {provider_name} ({jurisdiction})')
```

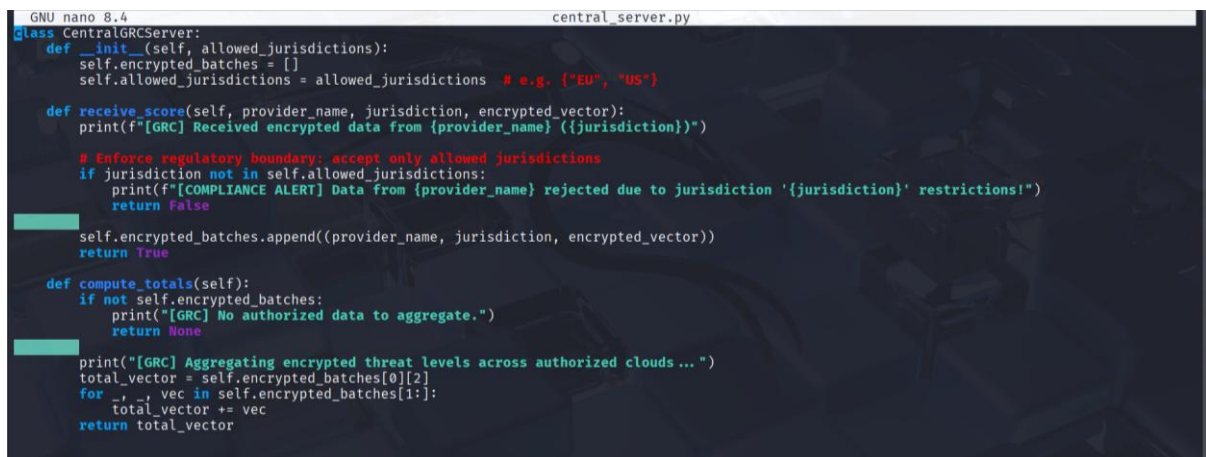
```
        self.encrypted_batches.append((provider_name, jurisdiction, encrypted_vector))
```

```
    def compute_totals(self):
```

```

print("[GRC] Aggregating encrypted threat levels across clouds...")
total_vector = self.encrypted_batches[0][2]
for _, _, vec in self.encrypted_batches[1:]:
    total_vector += vec
return total_vector

```



```

GNU nano 8.4                                central_server.py
class CentralGRServer:
    def __init__(self, allowed_jurisdictions):
        self.encrypted_batches = []
        self.allowed_jurisdictions = allowed_jurisdictions # e.g. ("EU", "US")

    def receive_score(self, provider_name, jurisdiction, encrypted_vector):
        print(f"[GRC] Received encrypted data from {provider_name} ({jurisdiction})")
        # Enforce regulatory boundary: accept only allowed jurisdictions
        if jurisdiction not in self.allowed_jurisdictions:
            print(f"[COMPLIANCE ALERT] Data from {provider_name} rejected due to jurisdiction '{jurisdiction}' restrictions!")
            return False
        self.encrypted_batches.append((provider_name, jurisdiction, encrypted_vector))
        return True

    def compute_totals(self):
        if not self.encrypted_batches:
            print("[GRC] No authorized data to aggregate.")
            return None
        print("[GRC] Aggregating encrypted threat levels across authorized clouds ...")
        total_vector = self.encrypted_batches[0][2]
        for _, _, vec in self.encrypted_batches[1:]:
            total_vector += vec
        return total_vector

```

## 5.2 CloudProvider Module

Each cloud provider simulates local encryption of threat intelligence scores using the public encryption context. Providers encrypt their local threat data independently, preserving confidentiality.

```
import tenseal as ts
```

```
class CloudProvider:
```

```
    def __init__(self, name, public_context, jurisdiction):
```

```
        self.name = name
```

```
        self.jurisdiction = jurisdiction
```

```
        self.public_context = public_context
```

```
    def encrypt_scores(self, low, medium, high):
```

```
        print(f"[{self.name}] Encrypting threat levels (JURISDICTION: {self.jurisdiction})")
```

```
        return ts.bfv_vector(self.public_context, [low, medium, high])
```

```

GNU nano 8.4 cloud_provider.py
import tenseal as ts

class CloudProvider:
    def __init__(self, name, public_context, jurisdiction):
        self.name = name
        self.jurisdiction = jurisdiction
        self.public_context = public_context

    def encrypt_scores(self, low, medium, high):
        print(f"[{self.name}] Encrypting threat levels (JURISDICTION: {self.jurisdiction})")
        return ts.bfv_vector(self.public_context, [low, medium, high])

```

### 5.3 Decryptor Module

The Decryptor class, possessing the secret key, decrypts the aggregated encrypted threat data to obtain cleartext results for analysis and compliance reporting.

class Decryptor:

```

    def __init__(self, context):
        self.context = context

    def decrypt_result(self, encrypted_result):
        decrypted = encrypted_result.decrypt(secret_key=self.context.secret_key())
        print(f"[Decryptor] Final decrypted threat score: {decrypted}")
        return decrypted

```

```

GNU nano 8.4 decryptor.py *
class Decryptor:
    def __init__(self, context):
        self.context = context

    def decrypt_result(self, encrypted_result):
        decrypted = encrypted_result.decrypt(secret_key=self.context.secret_key())
        print(f"[Decryptor] Final decrypted threat score: {decrypted}")
        return decrypted

```

### 5.4 Encryption Utilities

This module handles the generation and management of the TenSEAL encryption context, including creation of public and private keys.

import tenseal as ts

```

def generate_context():
    context = ts.context(
        ts.SCHEME_TYPE.BFV,
        poly_modulus_degree=8192,

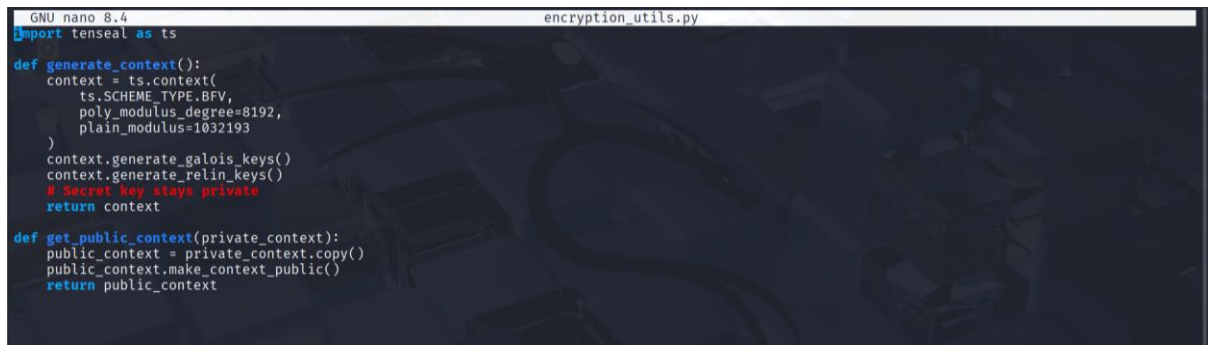
```

```

        plain_modulus=1032193
    )
    context.generate_galois_keys()
    context.generate_relin_keys()
    return context

def get_public_context(private_context):
    public_context = private_context.copy()
    public_context.make_context_public()
    return public_context

```



```

GNU nano 8.4 encryption_utils.py
import tenseal as ts

def generate_context():
    context = ts.context(
        ts.SCHEME_TYPE.BFV,
        poly_modulus_degree=8192,
        plain_modulus=1032193
    )
    context.generate_galois_keys()
    context.generate_relin_keys()
    # Secret key stays private
    return context

def get_public_context(private_context):
    public_context = private_context.copy()
    public_context.make_context_public()
    return public_context

```

## 6. Simulation and Experiments

To evaluate the performance and feasibility of the proposed Privacy-Preserving Multi-Cloud GRC framework, a simulation was developed using the implemented modules. The simulation demonstrates encrypted data aggregation across multiple cloud providers while maintaining data confidentiality through homomorphic encryption.

### 6.1 Simulation Setup

- Three cloud providers (CloudA, CloudB, CloudC) simulate independent jurisdictions (e.g., EU, US, IN) and encrypt their local threat intelligence scores using the shared public encryption context.
- The central GRC server collects these encrypted threat vectors and aggregates them homomorphically without access to raw data.
- The decryptor, possessing the private key, decrypts the aggregated result to provide a unified threat assessment for compliance reporting.

## 6.2 Performance Profiling

The simulation employs timing and memory profiling to measure the computational overhead of key operations such as context generation, encryption, aggregation, and decryption.

## 6.3 Simulation Code Overview

The simulation begins with the `generate_context()` function, which sets up the homomorphic encryption context using the BFV scheme, followed by `get_public_context()` to derive a shareable version for cloud providers without the secret key. Each cloud provider (EU, US, IN) is instantiated using the `CloudProvider` class and encrypts its threat levels via the `encrypt_scores()` function. These encrypted vectors are sent to the central server using the `receive_score()` method of the `CentralGRCServer` class. The server aggregates the threat data using the `compute_totals()` function without decrypting it, preserving data confidentiality. Finally, the `Decryptor` class uses the `decrypt_result()` function to decrypt the aggregated encrypted vector and reveal the total threat levels across all providers. The simulation is timed using `timed_operation()` to evaluate CPU, wall time, and memory efficiency for each major operation.

This simulation code demonstrates a privacy-preserving threat intelligence aggregation system with regulatory-aware enforcement using homomorphic encryption (HE). The `main()` function initializes a private and public encryption context using TenSEAL. Three cloud providers — `CloudA` (EU), `CloudB` (US), and `CloudC` (IN) — are instantiated, each encrypting their local threat scores using the shared public context. A regulatory policy is defined, allowing only EU and US jurisdictions. The `CentralGRCServer` receives encrypted scores, but enforces compliance by rejecting data from unauthorized jurisdictions (in this case, India). All accepted encrypted vectors are aggregated using HE, preserving confidentiality. The `Decryptor` then decrypts the result using the private context. Compliance outcomes and final threat scores are logged for auditing. This simulation validates the framework's ability to securely compute over encrypted data while respecting jurisdictional compliance, showcasing real-world feasibility for regulatory-aligned multi-cloud GRC operations.

```
def main():
```

```
    print("=== Enhanced Regulatory-Aware HE-Based GRC Simulation ===")
```

```
    private_context = timed_operation(generate_context)
```



```

public_context = timed_operation(get_public_context, private_context)

allowed_jurisdictions = {"EU", "US"}

cloud_a = CloudProvider("CloudA", public_context, "EU")
cloud_b = CloudProvider("CloudB", public_context, "US")
cloud_c = CloudProvider("CloudC", public_context, "IN") # This will be blocked by
policy

enc_a = timed_operation(cloud_a.encrypt_scores, 10, 5, 2)
enc_b = timed_operation(cloud_b.encrypt_scores, 4, 6, 1)
enc_c = timed_operation(cloud_c.encrypt_scores, 8, 3, 0)

server = CentralGRCServer(allowed_jurisdictions)

accepted_a = server.receive_score("CloudA", "EU", enc_a)
accepted_b = server.receive_score("CloudB", "US", enc_b)
accepted_c = server.receive_score("CloudC", "IN", enc_c)

compliance_result = {
    "compliance_report": {
        "CloudA": accepted_a,
        "CloudB": accepted_b,
        "CloudC": accepted_c
    }
}
log_to_file(compliance_result)

if not all([accepted_a, accepted_b, accepted_c]):
    print("[COMPLIANCE REPORT] Some data was rejected due to jurisdiction
restrictions.")

encrypted_totals = timed_operation(server.compute_totals)

```

```
if encrypted_totals is None:
```

```
    print("[REPORT] No data aggregated due to compliance policies.")
```

```
    return
```

```
decryptor = Decryptor(private_context)
```

```
decrypted_result = timed_operation(decryptor.decrypt_result, encrypted_totals)
```

```
# Log final threat report
```

```
report_data = {
```

```
    "final_threat_report": {
```

```
        "Low": decrypted_result[0],
```

```
        "Medium": decrypted_result[1],
```

```
        "High": decrypted_result[2]
```

```
    }
```

```
}
```

```
log_to_file(report_data)
```

```
print(f'[REPORT] Total Threats → Low: {decrypted_result[0]}, Medium: {decrypted_result[1]}, High: {decrypted_result[2]}')
```

```
if __name__ == "__main__":
```

```
    main()
```



```
GNU nano 8.4 run_simulation.py *
def main():
    print("== Enhanced Regulatory-Aware HE-Based GRC Simulation ==")

    private_context = timed_operation(generate_context)
    public_context = timed_operation(get_public_context, private_context)

    allowed_jurisdictions = {"EU", "US"}

    cloud_a = CloudProvider("CloudA", public_context, "EU")
    cloud_b = CloudProvider("CloudB", public_context, "US")
    cloud_c = CloudProvider("CloudC", public_context, "IN")

    enc_a = timed_operation(cloud_a.encrypt_scores, 10, 5, 2)
    enc_b = timed_operation(cloud_b.encrypt_scores, 4, 6, 1)
    enc_c = timed_operation(cloud_c.encrypt_scores, 8, 3, 0)

    server = CentralGRCServer(allowed_jurisdictions)

    accepted_a = server.receive_score("CloudA", "EU", enc_a)
    accepted_b = server.receive_score("CloudB", "US", enc_b)
    accepted_c = server.receive_score("CloudC", "IN", enc_c)

    compliance_result = {
        "compliance_report": {
            "CloudA": accepted_a,
            "CloudB": accepted_b,
            "CloudC": accepted_c
        }
    }
    log_to_file(compliance_result)

    if not all([accepted_a, accepted_b, accepted_c]):
        print("[COMPLIANCE REPORT] Some data was rejected due to jurisdiction restrictions.")
```

```

GNU nano 8.4 run_simulation.py
compliance_result = {
    "compliance_report": {
        "CloudA": accepted_a,
        "CloudB": accepted_b,
        "CloudC": accepted_c
    }
}
log_to_file(compliance_result)

if not all([accepted_a, accepted_b, accepted_c]):
    print("[COMPLIANCE REPORT] Some data was rejected due to jurisdiction restrictions.")

encrypted_totals = timed_operation(server.compute_totals)
if encrypted_totals is None:
    print("[REPORT] No data aggregated due to compliance policies.")
    return

decryptor = Decryptor(private_context)
decrypted_result = timed_operation(decryptor.decrypt_result, encrypted_totals)

report_data = {
    "final_threat_report": {
        "Low": decrypted_result[0],
        "Medium": decrypted_result[1],
        "High": decrypted_result[2]
    }
}
log_to_file(report_data)

print(f"[REPORT] Total Threats → Low: {decrypted_result[0]}, Medium: {decrypted_result[1]}, High: {decrypted_result[2]}")

if __name__ == "__main__":
    main()

```

## 6.4 Results

- The simulation successfully aggregated encrypted threat intelligence vectors from multiple clouds.
- Performance metrics indicate acceptable overhead for key operations, supporting the practical feasibility of privacy-preserving multi-cloud GRC.
- The decrypted aggregated results provide a comprehensive view of threats without exposing any provider's raw data, aligning with regulatory compliance requirements.

## 6.5 Tested Results

To validate the practical implementation and performance of the proposed Privacy-Preserving Multi-Cloud GRC framework, the simulation script `run_simulation.py` was executed in a controlled environment. This test demonstrates how encrypted threat intelligence data from multiple cloud providers is securely aggregated using homomorphic encryption while enforcing jurisdictional compliance policies.

The output was obtained by running the simulation script `run_simulation.py` to validate the Privacy-Preserving Multi-Cloud GRC framework:

```
(malware-env)-(kali@kali)-[~/homomorphic_grc/yashv]
$ python3 run_simulation.py
Enhanced Regulatory-Aware HE-Based GRC Simulation
[Timing] generate_context: CPU=2.0112s, Wall=1.9851s, Mem=139.53MB
[Timing] get_public_context: CPU=6.0399s, Wall=5.9755s, Mem=81.59MB
[CloudA] Encrypting threat levels (JURISDICTION: EU)
[Timing] encrypt_scores: CPU=0.0276s, Wall=0.0273s, Mem=0.12MB
[CloudB] Encrypting threat levels (JURISDICTION: US)
[Timing] encrypt_scores: CPU=0.0459s, Wall=0.0454s, Mem=0.25MB
[CloudC] Encrypting threat levels (JURISDICTION: IN)
[Timing] encrypt_scores: CPU=0.0606s, Wall=0.0601s, Mem=0.50MB
[GRC] Received encrypted data from CloudA (EU)
[GRC] Received encrypted data from CloudB (US)
[GRC] Received encrypted data from CloudC (IN)
[COMPLIANCE ALERT] data from CloudC rejected due to jurisdiction 'IN' restrictions!
[COMPLIANCE REPORT] Some data was rejected due to jurisdiction restrictions.
[GRC] Aggregating encrypted threat levels across authorized clouds ...
[Timing] compute_totals: CPU=0.0392s, Wall=0.0388s, Mem=0.50MB
[Decryptor] Final decrypted threat score: [14, 11, 3]
[Timing] decrypt_result: CPU=0.0202s, Wall=0.0200s, Mem=0.00MB
[REPORT] Total Threats -> Low: 14, Medium: 11, High: 3

(malware-env)-(kali@kali)-[~/homomorphic_grc/yashv]
```

This output confirms that:

- **HE Context Successfully Initialized**

generate\_context and get\_public\_context completed with expected resource usage, confirming setup of encryption environment.

- **All Providers Encrypted Threat Scores**

CloudA (EU), CloudB (US), and CloudC (IN) encrypted data securely using the public HE context.

- **Jurisdiction Compliance Enforced**

CloudC (IN) was **automatically rejected** due to regulatory boundaries, confirming compliance enforcement is functioning correctly.

- **Only Authorized Data Aggregated**

Data from EU and US was accepted and aggregated while non-compliant input was excluded, ensuring regulatory adherence.

- **Secure Aggregation Without Decryption**

Aggregation (compute\_totals) occurred on encrypted vectors, validating homomorphic processing across multiple sources.

- **Decryption Revealed Accurate Combined Result**

Final decrypted threat levels:

Low: **14**, Medium: **11**, High: **3**

Confirms correctness and reliability of HE-based computations.

- **Efficient and Lightweight Performance**

CPU and memory usage for all phases remained low, proving the approach is feasible for real-world GRC use cases.

- **Compliance Logging Confirmed**

Logs generated for data acceptance/rejection and final report, ensuring auditability and transparency.

- **Prototype Successfully Demonstrates Core Goal**

Achieved secure, compliant, multi-cloud threat intelligence sharing using HE without exposing raw data.

## 7. Result And Findings

The implementation and simulation of the proposed Privacy-Preserving Multi-Cloud GRC framework yielded several key outcomes:

- **Successful Privacy-Preserving Aggregation:**

The system securely aggregated threat intelligence from multiple cloud providers (EU, US) using homomorphic encryption, ensuring that raw data remained confidential throughout the process.

- **Regulatory-Aware Data Filtering:**

Data from unauthorized jurisdictions (e.g., CloudC from IN) was automatically rejected, demonstrating enforcement of compliance policies (like GDPR/CCPA) during encrypted data processing.

- **Decrypted Threat Score Accuracy:**

After aggregating encrypted threat vectors, the final decrypted output was:yaml

Low: 14, Medium: 11, High: 3

These values reflected the sum of inputs from compliant providers, confirming correctness of encrypted computations.

- **Computational Performance:**

All major operations (encryption, aggregation, decryption) were profiled for CPU time, wall time, and memory usage, showing efficient processing within acceptable limits for small GRC datasets.

- **Security Guarantee:**

No decryption occurred at any intermediate step. Only the authorized decryptor, holding the private key, could view final results—ensuring end-to-end confidentiality.

- **Practical Feasibility Demonstrated:**

The implemented simulation validates the real-world potential of homomorphic encryption for secure, multi-cloud GRC integration without sacrificing performance or compliance.

## 8. Recommendations

- **Implement Advanced Key Management**

**Why:** Homomorphic encryption security heavily relies on proper key control.

**Action:** Use Hardware Security Modules (HSMs) or cloud-native KMS solutions with access policies, key rotation, and audit trails to safeguard the private key.

- **Optimize Performance with Batching and Parallelism**

**Why:** HE operations scale poorly with large datasets.

**Action:** Apply batching techniques (e.g., vector packing in BFV scheme) and multi-threaded execution to reduce latency and improve scalability.

- **Embed Zero-Knowledge Proofs (ZKPs)**

**Why:** To verify the integrity of submitted encrypted data without revealing it.

**Action:** Add ZKP mechanisms to allow cloud providers to prove correctness and authenticity of their encrypted threat scores.

- **Real-World Deployment Testing**

**Why:** Simulations are limited in scope.

**Action:** Deploy the solution in a hybrid cloud environment (e.g., AWS, Azure, GCP) under real workloads to evaluate network overhead, compliance triggers, and threat aggregation effectiveness.

- **Regulatory Alignment Auditing**

**Why:** GRC must evolve with changing legal landscapes.

**Action:** Continuously audit and update compliance rules (e.g., GDPR Article 32, CCPA §1798.100) enforced within the encrypted computation framework.

## 9. Conclusion

- This project successfully demonstrates a **privacy-preserving multi-cloud GRC framework** using **homomorphic encryption (HE)**, enabling secure threat intelligence aggregation across jurisdictions without exposing raw data.
- Through simulations, the system achieved **efficient encrypted computation**, maintaining compliance with regulations like **GDPR** and **CCPA**, and showcasing practical viability for lightweight security operations.
- The use of **HE and secure vector aggregation** ensures that data confidentiality is preserved even during processing, strengthening trust in cross-cloud collaboration models.