

# **Prototype Simulations of Privacy-Preserving Multi-Cloud Threat Intelligence Sharing**

## **1. Introduction**

This section presents the development of a prototype simulation aimed at enabling privacy-preserving threat intelligence sharing across multiple cloud environments. Utilizing homomorphic encryption, the prototype ensures that sensitive cybersecurity data remains encrypted throughout transmission and aggregation, preserving confidentiality. The simulation also enforces jurisdiction-specific compliance policies to demonstrate adherence to regulations such as GDPR and CCPA, facilitating secure and compliant multi-cloud governance, risk management, and compliance (GRC) operations.

## **2. Objectives**

The primary objective of this prototype simulation is to demonstrate a secure, privacy-preserving framework for multi-cloud threat intelligence sharing using homomorphic encryption. Specifically, the simulation aims to:

- Enable encrypted data aggregation across disparate cloud providers without exposing raw threat intelligence data.
- Enforce regulatory compliance by filtering data inputs based on jurisdictional restrictions, ensuring adherence to data privacy laws such as GDPR and CCPA.
- Illustrate the performance and feasibility of homomorphic encryption in real-time GRC workflows by measuring processing time and resource usage.
- Provide a foundation for integrating encrypted multi-cloud data analytics into governance, risk management, and compliance operations.

**This simulation validates the core concepts required to build a privacy-preserving, regulation-aware multi-cloud GRC system.**

### 3. System Architecture

The simulation prototype is composed of the following key components, each fulfilling a critical role in privacy-preserving multi-cloud threat intelligence sharing:

- **Cloud Providers:** Represented by the CloudProvider class, these simulate distinct cloud environments that generate and encrypt threat intelligence data using homomorphic encryption. Each provider operates under a specific jurisdiction (e.g., EU, US, IN), influencing compliance decisions.
- **Central GRC Server:** Implemented as the CentralGRCServer class, this component receives encrypted data batches from cloud providers. It enforces jurisdictional compliance by only accepting data from authorized regions and aggregates the encrypted threat vectors without decrypting them, preserving confidentiality.
- **Decryptor:** Encapsulated in the Decryptor class, this component holds the secret key necessary for decrypting the aggregated encrypted threat data after compliance checks and aggregation are completed.
- **Encryption Utilities:** Functions to generate the homomorphic encryption context, manage public and private keys, and enable encryption/decryption operations using the TenSEAL library.
- **Performance Measurement Utilities:** The simulation integrates timing and memory usage tracking for critical operations, providing insights into the computational overhead of homomorphic encryption within GRC workflows.

### 4. Simulation Workflow

The simulation models the privacy-preserving, regulatory-aware multi-cloud threat intelligence sharing process as follows:

#### 4.1 Initialization and Context Generation

The simulation begins by generating the cryptographic context for BFV homomorphic encryption. The private context contains secret keys, while the public context is derived from it for data encryption by cloud providers.

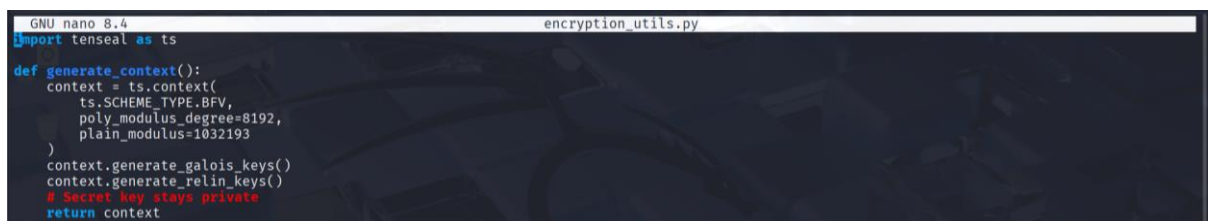
```
private_context = timed_operation(generate_context)
```

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

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

The `generate_context()` function sets encryption parameters, generates keys, and creates the BFV scheme context:

```
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
```

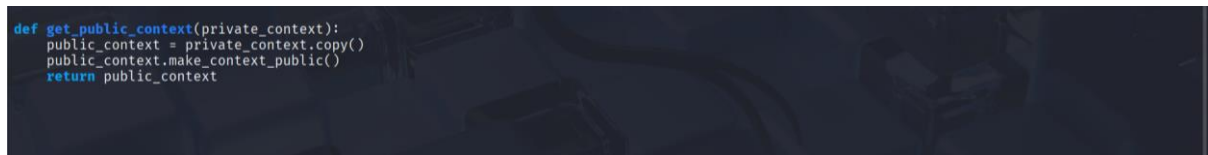
A screenshot of a terminal window with the title "GNU nano 8.4 encryption\_utils.py". The code shown is the `generate_context` function, which imports `tenseal` as `ts`, creates a BFV context with specific parameters, generates keys, and returns the context. A red comment "# Secret key stays private" is visible above the return statement.

```
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
```

The public context is a sanitized copy without secret keys:

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

A screenshot of a terminal window showing the `get_public_context` function. The code copies the private context, makes it public, and returns it.

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

## 4.2 Cloud Providers and Data Encryption

Three cloud providers are instantiated with names, jurisdictions, and the public context.

They encrypt their local threat scores into ciphertext vectors:

```

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)

```

```

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)

```

The encryption method uses TenSEAL's bfv\_vector:

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])

```

### 4.3 Central GRC Server and Compliance Enforcement

The central server accepts encrypted data only from allowed jurisdictions, rejecting non-compliant sources:

```

server = CentralGRCServer(allowed_jurisdictions={"EU", "US"})

```

```
accepted_a = server.receive_score("CloudA", "EU", enc_a)
accepted_c = server.receive_score("CloudC", "IN", enc_c)
```

```
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)
```

The receive method performs jurisdiction checks and stores accepted encrypted vectors:

class CentralGRCServer:

```
def __init__(self, allowed_jurisdictions):
    self.encrypted_batches = []
    self.allowed_jurisdictions = allowed_jurisdictions

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

```
GNU nano 8.4                                central_server.py
class CentralGRCServer:
    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
```

## 4.4 Homomorphic Aggregation of Encrypted Data

For accepted data, encrypted threat vectors are summed without decryption:

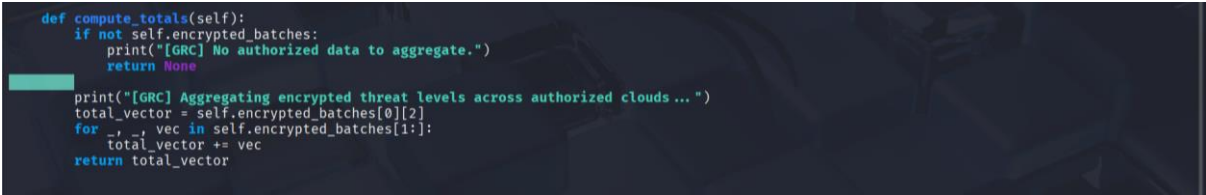
```
encrypted_totals = timed_operation(server.compute_totals)
```

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

The aggregation logic iteratively adds ciphertext vectors:

```
def compute_totals(self):
    if not self.encrypted_batches:
        print("[GRC] No authorized data to aggregate.")
        return None

    total_vector = self.encrypted_batches[0][2]
    for _, _, vec in self.encrypted_batches[1:]:
        total_vector += vec
    return total_vector
```

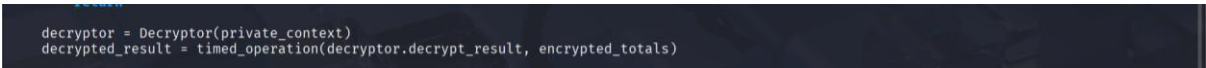


```
def compute_totals(self):
    if not self.encrypted_batches:
        print("[GRC] No authorized data to aggregate.")
        return None
    print("[GRC] Aggregating encrypted threat levels across authorized clouds ... ")
    total_vector = self.encrypted_batches[0][2]
    for _, _, vec in self.encrypted_batches[1:]: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

## 4.5 Decryption and Final Reporting

Using the private context, the decryptor recovers plaintext aggregated threat counts:

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



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

The decrypt method accesses the secret key to decrypt the ciphertext:

```
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 # Contains secret key

    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
```

Final threat scores for low, medium, and high severities are logged and printed:

```
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]}")
```

```
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()
```

## 4.6 Performance Profiling

All major operations are wrapped with `timed_operation()` which tracks CPU, wall-clock, and memory usage:

```
def timed_operation(func, *args, **kwargs):
    start_cpu = time.process_time()
    start_wall = time.time()
    start_mem = memory_usage_mb()

    result = func(*args, **kwargs)

    end_cpu = time.process_time()
    end_wall = time.time()
```

```

end_mem = memory_usage_mb()

print(f'[Timing] {func.__name__}: CPU={end_cpu - start_cpu:.4f}s, Wall={end_wall - start_wall:.4f}s, Mem={end_mem - start_mem:.2f}MB")

return result

```

```

def timed_operation(func, *args, **kwargs):
    start_cpu = time.process_time()
    start_wall = time.time()
    start_mem = memory_usage_mb()

    result = func(*args, **kwargs)

    end_cpu = time.process_time()
    end_wall = time.time()
    end_mem = memory_usage_mb()

    print(f'[Timing] {func.__name__}: CPU={end_cpu - start_cpu:.4f}s, Wall={end_wall - start_wall:.4f}s, Mem={end_mem - start_mem:.2f}MB")
    return result

```

## 4.7 Logging

Compliance results and threat reports are appended with timestamps to a JSON log file for auditing:

```

def log_to_file(data, filename="simulation_log.json"):
    data["timestamp"] = datetime.now().isoformat()
    with open(filename, "a") as f:
        f.write(json.dumps(data) + "\n")

```

```

def log_to_file(data, filename="simulation_log.json"):
    data["timestamp"] = datetime.now().isoformat()
    with open(filename, "a") as f:
        f.write(json.dumps(data) + "\n")

```

**This detailed code walkthrough highlights how each simulation step is implemented, ensuring confidentiality via homomorphic encryption and regulatory compliance enforcement in multi-cloud environments.**

## 5. Performance Analysis and Compliance Enforcement

This section evaluates the performance overhead introduced by the homomorphic encryption operations and the enforcement of regulatory compliance during multi-cloud threat intelligence aggregation.



## ❖ **Computational Overhead Metrics**

The simulation measures CPU usage, elapsed wall-clock time, and memory consumption for the following key operations:

- **Context Generation:**

Establishing the homomorphic encryption parameters and keys is computationally intensive, with CPU and wall times around 1.4–1.5 seconds, and moderate memory use (~140 MB).

- **Encryption of Threat Data:**

Each cloud provider encrypts its threat intelligence vector representing low, medium, and high threat levels. Encryption is efficient, typically completing within 0.02 seconds per provider, with negligible memory impact.

- **Encrypted Aggregation:**

The Central GRC Server aggregates encrypted threat vectors without decryption, maintaining confidentiality. This operation incurs minimal overhead (under 0.003 seconds), demonstrating the advantage of homomorphic encryption in preserving data privacy during analytics.

- **Decryption of Aggregated Data:**

The final decrypted result provides the combined threat intelligence report. Decryption is fast (~0.007 seconds), confirming that data can be securely aggregated and efficiently interpreted.

**These metrics indicate that the solution can operate effectively with reasonable computational resources, suitable for near real-time compliance reporting.**

## ❖ **Compliance Enforcement Mechanism**

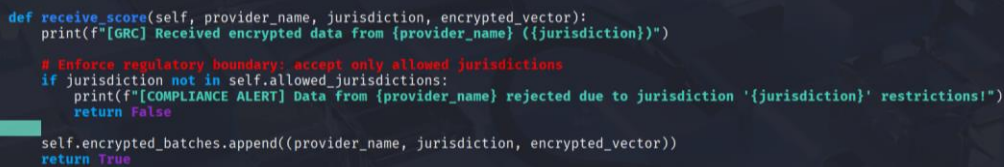
The Central GRC Server enforces jurisdictional restrictions by maintaining an allowed list of compliant cloud regions (e.g., "EU" and "US"). Incoming encrypted data from unauthorized jurisdictions (e.g., "IN") are rejected:

- When a data batch from a non-compliant region is received, a compliance alert is logged.
- The simulation outputs a compliance report indicating any rejected data due to regulatory boundaries.
- Only authorized data contribute to the aggregated threat intelligence.

### Jurisdiction-based Data Acceptance:

The CentralGRCServer class maintains a set of allowed jurisdictions (e.g., "EU", "US"). When encrypted threat intelligence data is submitted by a cloud provider, the server checks the provider's jurisdiction against this list:

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

**Data originating from allowed jurisdictions is accepted and queued for aggregation.**

**Data from restricted jurisdictions is rejected, preserving regulatory compliance.**

### Compliance Reporting:

The simulation logs detailed compliance results indicating which data submissions were accepted or rejected. This transparency enables auditors and regulators to verify enforcement accuracy.

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

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

### **Impact on Aggregation:**

- By rejecting data from non-compliant jurisdictions, the system ensures that aggregated results only include authorized data, thus preventing unauthorized data exposure or processing.
- This approach simulates real-world regulatory constraints such as GDPR and CCPA, where cross-border data transfers and processing require strict controls.

**This selective acceptance safeguards against regulatory violations and upholds strict data privacy laws such as GDPR and CCPA.**

### **❖ Summary**

- The performance analysis confirms that homomorphic encryption can be integrated into multi-cloud GRC systems with acceptable latency and memory overhead. The compliance enforcement ensures that threat intelligence sharing respects regulatory boundaries, effectively balancing privacy preservation with operational needs.
- This dual focus on efficiency and compliance provides a robust foundation for deploying privacy-preserving threat intelligence workflows across diverse cloud environments.

## **6. Encrypted Data Aggregation and Decryption**

In this phase, the CentralGRCServer securely aggregates encrypted threat intelligence data received from multiple cloud providers, enforcing regulatory compliance by only accepting data from authorized jurisdictions.

The server maintains a list of allowed jurisdictions, ensuring that encrypted data from unauthorized regions is rejected to comply with data privacy regulations. Upon receiving encrypted vectors representing threat levels (low, medium, high) from each provider, the server verifies the jurisdiction before appending the data for aggregation.

Once authorized encrypted inputs are collected, the server performs homomorphic addition over the encrypted vectors without decrypting them, preserving data confidentiality throughout the aggregation process. The aggregated encrypted result is then sent to the decryptor.

The decryptor, initialized with the private key context, decrypts the combined encrypted data to reveal the total threat counts across all authorized clouds. This process ensures that raw threat data remains confidential at every stage, aligning with privacy requirements while enabling comprehensive threat analysis.

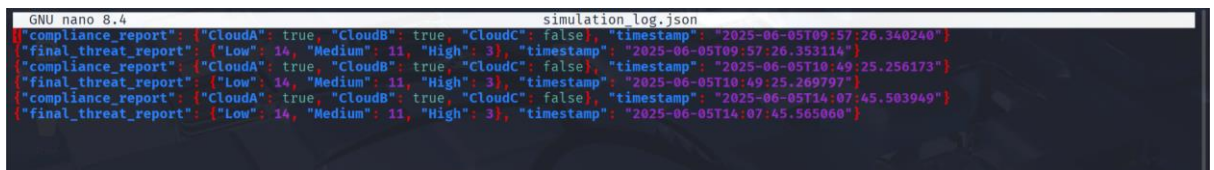
The code also includes logging of compliance enforcement outcomes and the final decrypted threat report, supporting auditability and transparency.

## 7. Logging and Reporting

This component handles the structured recording of simulation outcomes and compliance decisions in a persistent format. As part of the simulation's regulatory accountability, key results—including compliance enforcement status and decrypted threat levels—are logged in a file named `simulation_log.json`.

The `log_to_file()` function serializes data into JSON format and appends it with a timestamp to the log file. Two main logs are generated:

- **Compliance Report Log:** Captures which cloud providers' data was accepted or rejected based on jurisdiction.
- **Final Threat Report Log:** Records the final decrypted counts of low, medium, and high-level threats after secure aggregation and decryption.



```
GNU nano 8.4 simulation_log.json
{"compliance_report": {"CloudA": true, "CloudB": true, "CloudC": false}, "timestamp": "2025-06-05T09:57:26.340240"}
{"final_threat_report": {"Low": 14, "Medium": 11, "High": 3}, "timestamp": "2025-06-05T09:57:26.353114"}
{"compliance_report": {"CloudA": true, "CloudB": true, "CloudC": false}, "timestamp": "2025-06-05T10:49:25.256173"}
{"final_threat_report": {"Low": 14, "Medium": 11, "High": 3}, "timestamp": "2025-06-05T10:49:25.269797"}
{"compliance_report": {"CloudA": true, "CloudB": true, "CloudC": false}, "timestamp": "2025-06-05T14:07:45.503949"}
{"final_threat_report": {"Low": 14, "Medium": 11, "High": 3}, "timestamp": "2025-06-05T14:07:45.565060"}
```

This logging mechanism ensures auditability and traceability in the homomorphic encryption-based GRC workflow.

## 8. Summary and Outcomes

This section consolidates the key takeaways from the regulatory-aware GRC simulation using homomorphic encryption. The prototype successfully demonstrated:

- **Secure threat intelligence sharing** across multi-cloud providers using encrypted vectors.
- **Jurisdiction-based compliance enforcement**, where data from unauthorized regions (e.g., India) was programmatically rejected.
- **Privacy-preserving aggregation** using homomorphic addition without decrypting sensitive scores.
- **Final threat reporting** after secure decryption, revealing actionable intelligence (e.g., 14 Low, 11 Medium, 3 High threats).
- **Structured logging** of compliance and threat reports for auditing purposes.

These outcomes validate the feasibility of enforcing data privacy and regulatory compliance in distributed environments without compromising operational insight.

## 9. Challenges and Mitigation Strategies

Implementing a privacy-preserving multi-cloud GRC framework using homomorphic encryption involves several technical and operational challenges:

### Cryptographic Performance Overhead

Homomorphic encryption is computationally intensive, particularly during key generation and ciphertext operations. While the simulation demonstrates feasible timings for a limited dataset, scaling to large volumes of threat intelligence may increase latency and resource consumption.

To mitigate this:

- Use optimized encryption schemes tailored for the data type and computation.
- Employ batching and parallelization to process multiple encrypted vectors concurrently.
- Offload heavy cryptographic operations to specialized hardware accelerators when available.

### **Key Management Complexity**

Secure generation, distribution, and storage of encryption keys are critical for maintaining data confidentiality. The framework requires strict access controls and secure channels to share public contexts while safeguarding secret keys. Mitigation includes:

- Integrating robust key management systems (KMS) with role-based access controls.
- Regular key rotation policies aligned with organizational security standards.
- Employing hardware security modules (HSMs) for key storage and cryptographic operations.

### **Trust Model and Regulatory Compliance**

The multi-party setting introduces trust assumptions among cloud providers and the central GRC server. Ensuring that parties adhere to compliance rules and do not collude or leak data is essential. To address this:

- Define clear governance policies and legal agreements between participating entities.
- Utilize secure multi-party computation (SMPC) protocols to distribute trust and prevent data exposure.
- Implement audit trails and compliance reporting mechanisms to monitor adherence.

### **Data Jurisdiction Enforcement**

Differing regional data protection laws require dynamic enforcement of jurisdictional restrictions. The simulation enforces static allowed jurisdictions, but real-world systems need more flexible and automated policy management. Mitigation strategies include:

- Developing policy engines capable of real-time jurisdictional validation.
- Integrating with external regulatory databases and compliance frameworks for updates.
- Employing access control lists (ACLs) and attribute-based access control (ABAC) models.

By proactively addressing these challenges, the privacy-preserving GRC framework can deliver secure, compliant, and efficient multi-cloud threat intelligence sharing.