SplitSecure: Breaking the Monolith for Crypto Agility

Abstract

As quantum computing advances, modern cryptographic protocols such as RSA and ECDSA face the risk of becoming obsolete, threatening the security of high-stakes financial systems like India's Unified Payments Interface (UPI). This paper proposes a comprehensive migration strategy toward post-quantum cryptography (PQC) by introducing a crypto-agile microservice architecture integrated with enhanced Hardware Security Module (HSM) support. The solution decouples cryptographic operations from application logic, enabling secure, scalable, and flexible cryptography updates. This paper mainly aims to include the implementation of a hybrid TLS stack supporting classical and post-quantum key exchanges, JWT token signing using algorithms like Dilithium and Falcon, hybrid certificate support for backward compatibility, and a functional proof-of-concept using SoftHSM with liboqs over PKCS#11. This architecture allows dynamic cryptographic policy enforcement, ensures quantum resistance, and maintains operational continuity during cryptographic transitions. Our work offers a blueprint for secure post-quantum adoption in financial ecosystems, ensuring resilience against both current and future cryptographic threats.

2. Introduction

2.1 Post-Quantum Cryptography

With the rapid technological strides in quantum computing, it is increasingly likely that quantum systems capable of breaking widely used public-key cryptography will emerge within the next two decades. Unlike classical computers, which require infeasible amounts of time to brute-force algorithms like AES-256, quantum computers harness principles such as superposition and entanglement to solve certain problems exponentially faster. Shor's algorithm, for instance, can factor large numbers and compute discrete logarithms in polynomial time—breaking RSA and ECDSA, the cornerstones of secure internet communication today.

In anticipation of this threat, a new field of cryptography has emerged: Post-Quantum Cryptography (PQC). PQC refers to cryptographic algorithms designed to resist quantum attacks while maintaining classical security. These algorithms must be mathematically complex enough to withstand both classical and quantum adversaries. Institutions like NIST and ISO are spearheading the effort to evaluate and standardize these algorithms for real-world deployment.

2.2 Problem Statement

Modern financial systems such as UPI rely on monolithic architectures with deeply embedded cryptographic logic. These systems are neither crypto-agile nor quantum-resistant. Transitioning to PQC in such infrastructures poses several challenges:

- Replacing RSA/ECDSA with PQC alternatives demands changes across the application stack.
- Monolithic cryptographic services hinder modular upgrades.
- Centralized HSMs, while secure, lack flexibility and PQC-readiness.

There is a pressing need for a crypto-agile, modular framework that can integrate PQC incrementally while ensuring backward compatibility, scalability, efficiency and compliance.

2.3 Current Scenario

India's UPI and associated platforms like PhonePe, Google Pay, and Paytm use TLS for secure transmission and JWTs for stateless authentication. These rely on classical algorithms vulnerable to quantum attacks. Current HSMs support only conventional key types and operate as tightly coupled modules within application stacks.

NIST has already finalized PQC algorithms such as **CRYSTALS-Kyber** (KEM) and **CRYSTALS-Dilithium**/**Falcon** (signatures), and real-world implementations have begun at organizations like Cloudflare and Cisco. However, financial systems still lack a tested migration path suitable for high-throughput, regulated environments.

2.4 Contributions

This paper presents a novel architecture that addresses these gaps by:

- Modularizing cryptographic operations into scalable microservices.

- Enabling PQC operations via hybrid TLS, JWT signing, and certificate issuance while maintaining crypto agility.
- Demonstrating practical integration through a SoftHSM + liboqs + PKCS#11 proof-of-concept.
- Proposing a crypto policy-driven, containerized deployment model for agility.

By decoupling cryptography from application logic and using a microservice paradigm, we provide a resilient migration blueprint for UPI-scale systems in the quantum era.

3. Literature Review

3.1 PQC Landscape

The National Institute of Standards and Technology (NIST) began standardizing PQC algorithms in 2016 and announced the first set of finalists in 2022. Among these are:

- **Kyber** – a lattice-based KEM (Key Encapsulation Mechanism)
- **Dilithium** and **Falcon** – lattice-based digital signature schemes
- **SPHINCS+** – a stateless hash-based signature scheme

These algorithms provide quantum resilience while supporting reasonable performance. Kyber768 and Dilithium3 are poised to be widely adopted in production systems due to their balance of size, speed, and security.

Table 3.1: Comparison of PQC Algorithms based on various factors.

Algorithm	Use	PubKey	PrivKey	Cipher/Sign Size	Speed (Sign/Enc)	Resource
Kyber-768	KEM (KeyEx)	~1.2KB	~2.4KB	~1.1KB (ctext)	Fast (ms)	Low
Dilithium3	Signature	~1.4KB	~3KB	~2.7KB (sig)	Moderate (ms)	Medium
Falcon-512	Signature	~0.9KB	~2.5KB	~0.7KB (sig)	Fast (ms)	Medium
SPHINCS+-256f	Signature	64B	128B	~35KB (sig)	Very Slow (s)	High

3.2 Cryptographic Practices in UPI Ecosystems

Current UPI systems typically employ:

- Monolithic TLS setups with classical X.509 certificates.
- Application-level JWT signing using libraries like PyJWT or Java JWT.
- Centralized HSMs accessed via PKCS#11 for secure key handling.

These setups offer limited agility and require significant rewrites to accommodate PQC algorithms. The absence of modular boundaries further complicates secure upgrades.

3.3 Related Work

Multiple research efforts and industry experiments have explored cryptographic agility and PQC readiness:

- **Cloudflare (2022)** deployed hybrid TLS handshakes using Kyber and X25519.
- **Cisco and ISARA** developed hybrid certificates combining classical and PQ signatures.
- **Samsung and Microsoft** have explored software-defined cryptography and microservice-based key management for post-quantum systems.
- **CARAF framework (2023)** introduces a risk assessment model for transitioning to cryptographic agility, emphasizing modular configuration and policy enforcement.

Academic literature increasingly highlights the need for **modular, containerized cryptography** that can evolve with emerging threats—a core principle of this paper's architecture.

4. Hardware Configuration

4.1 *Control and Coordination*

Main control unit (MCU), the main controller is used to keep track and perform high level functions. Routes and streamlines the operation to correct module based on the scheme as well as the operation to be done such as NTT multiplication or modular arithmetic.

Unified Polynomial Control Unit (UPCU), hardware unit to manage polynomial task and memory addressing.

While **Finite State Machine(FSM)** is a mathematical model system that can be in one of the finite states at any given time. It also controls NTT pipeline and internal communication.

TaPaSCo Framework- Task Parallel System Composer is an open-source framework on FPGA(a microprocessor chip). Helps to create, schedule, and run multiple hardware accelerators on a single chip easily and efficiently.
Components:-

- **SHELL** - that has control logic that handles I/O, resets, interrupts, and PE scheduling.
- **Processing Elements** - hardware accelerators like JPAU, KAM, NTT engine.
- **DMA Controller** - Direct memory access engine that sends/receives data from the host.
- **PCIe –** is used to bridge communication between the host computer and FPGA and other external devices like GPUs etc.

4.2 *Core Computation Modules*

Joint Polynomial Arithmetic Unit(JPAU) The main math engine that handles full polynomial operations — including NTT, inverse NTT, pointwise multiplication, and modular reduction. Focuses on performing arithmetic operations on **polynomials modulo a polynomial**.

Number Theoretic transform (NTT) Is a mathematic techniques, used for **large modular multiplication** operation. While the Fourier Transform uses sinusoidal functions (sine/cosine) in the complex field, the Number Theoretic Transform (NTT) operates entirely within a finite field using modular arithmetic on polynomials. Initially RSA and ECC were used and hence didn't require polynomial multiplication. But with PQC being implemented the large polynomial multiplication needs to be handled and processed. The input is converted to NTT , performed operation and converted to result using inverse NTT.

Butterfly Unit (BFU) is a hardware processing unit where data processed and flows in a pattern demonstrating flap of a butterfly where one value goes up while the other down. Designed using hardware gates. Only 2 inputs are possible at a time. Handles addition and subtraction with the ***TWIDDLE FACTOR***.

Montgomery multiplication – is used to reduce **modular repeated division with a prime modulo** within a finite field.

Polynomial multiplication - The high-level operation of multiplying two polynomials, crucial in PQC for key generation and encryption.

Modular Arithmetic Block - A hardware unit that performs fast addition, subtraction, or multiplication under a fixed modulus (mod q), used repeatedly during cryptographic computations.

JPAU = complete engine (orchestrator)
NTT = math method (a function/procedure)
BFU = basic compute unit (hardware block)

4.3 *Memory & Data Routing*

TWIDDLE Factor ROM is a precomputed constant, roots of unity required at each stage of NTT multiplication. It reduces recompilation time and increase efficiency. Stored is ROM. Fetched by BFU according to the input scheme.

Address/ memory mapper is a control logic unit. Control how memory is accessed and the pattern used to access. It calculates from where in BRAM the coefficient is stored in order to read from the memory.

Using dual port BRAM helps in speedy read/write process. **STRIDE Doubling per stage** ,the pattern in which at each stage the distance between 2 elements being combined gets doubled to prevent memory access conflict.

Shared RAM memory pool (SRAM) the central memory shared by all, used to store/hold **polynomials, keys, intermediate data**.

Dual Port BRAM Block RAM, FPGA memory that supports simultaneous read/write — useful for pipelined designs.

Pipeline Registers holds **intermediate results** at each NTT stage. Allows for pipeline processing in parallel.

4.4 *Hashing and Randomness*

KAT (Know Answer Test) in NIST is used to verify the hardware accelerator and its efficiency. This can be used by hardware vendors at the verification step to check the hardware configuration.
KAM (Keccak Acceleration Module) is used to speed up the hashing operation on SHAKE 128/256. Required for verification or signing digital signatures.
RNG (Random Number Generator) in Linux a function called randombytes() is used to generate the random number required for generating shared secret or a key and encapsulation.

4.5 *Hardware specific*

Table 4.1 FPGA Primitives and Development Tools Used in the Combined PQC Hardware Accelerator

Component	FPGA Primitive / Tool	Example Version / Device
HLS	Vivado HLS	Xilinx Vivado 2020.2+
HDL	Verilog / VHDL	N/A
Radix-2 NTT	BFU structure + $\log_2(n)$ stages	Implemented on Artix-7
DSP Blocks	DSP48E1 slices	Xilinx Artix-7 / Virtex-7
LUTs	6-input LUTs	Artix-7 family (e.g., XC7A200T)

Working of the Unified PQC Accelerator Architecture (Aligned with Block Diagram)

1. Task Initialization and Input Transfer - The host assembles all required data i.e.(plain text, message, public/private key, RNG buffer, scheme and mode). The complete input is sent to FPGA via PCIe through DMA.

2. Task Scheduling and Resource Allocation (TaPaSCo Shell) – The data is received by the TaPaSCo shell and it then interprets this incoming job. The internal FSM schedules task, selects appropriate hardware resources and initializes the UPCU.

3. Top-Level Control Flow - The **Main Control Unit (MCU)** decodes task metadata and algorithm selection bits. Based on the scheme:

- In Kyber/Dilithium/Falcon it activates **UPCU**
- In SPHINCS+ / hash-related tasks it transports data to the **Keccak Acceleration Module (KAM)**

4. Unified Polynomial Control Unit (UPCU) – The central controller unit for polynomial operation sends control signals to the following components :-

Butterfly Unit to start modular arithmetic.
Modular Arithmetic Block and **Montgomery Multiplier** and then to **Dual-Port BRAM** to perform read/write on coefficients. It also communicates with FSM to serialize NTT and inverse NTT stages.

5. Polynomial Arithmetic via JPAU Subsystem
Butterfly Unit (BFU) - The implementation involves stage-wise coefficient-wise operations, including addition, subtraction, and multiplication with twiddle factors, while efficiently operating in parallel configurations (2/4/8-way) tailored to the design scale. Then the Twiddle Factor ROM provides precomputed constants i.e roots of unity mod q, which are accessed systematically depending upon the NTT level during processing. The Modular Arithmetic Block performs all operations such as $(a \pm b) \bmod q$ or $(a \times b) \bmod q$, leveraging LUTs and DSP blocks for maximum efficiency. Additionally, the use of a Montgomery Multiplier, a specialized multiplier that effectively reduces the repeated division, ensures that operands are converted into the Montgomery domain, which results in significantly faster modular multiplication.

6. Memory Management and Data Routing
Dual-Port BRAM – Dual port permits concurrent read/write access while it also stores intermediated data of NTT domain as well as final result of NTT inverse. It also enables data level parallelism, dynamically calculates access indices to the BRAM and reduce stalling in Address Mapper, preventing memory conflict while access for read/write.

7. Keccak Acceleration Module (KAM) - Activated for schemes that require hashing, including Dilithium and SPHINCS+. This system efficiently accepts input from RNG for entropy and KAT vectors for deterministic validation. It delivers robust outputs, providing hash values for signature binding and intermediate outputs, ensuring seamless integration into the final format unit.

8. Output Finalization and Scheme-Specific Logic
Post-computation results are routed to the **Supported PQC Schemes Block**. This block formats outputs based on the scheme:

- Kyber: compresses ciphertext
- Dilithium: concatenates signature with message
- SPHINCS+: finalizes tree hash outputs

9. Verification

- Output is sent back to the host via PCIe (using DMA).
- The host compares the result with pre-loaded **KAT vectors** for correctness.
- If valid, the system is ready for the next job — no FPGA reprogramming required.

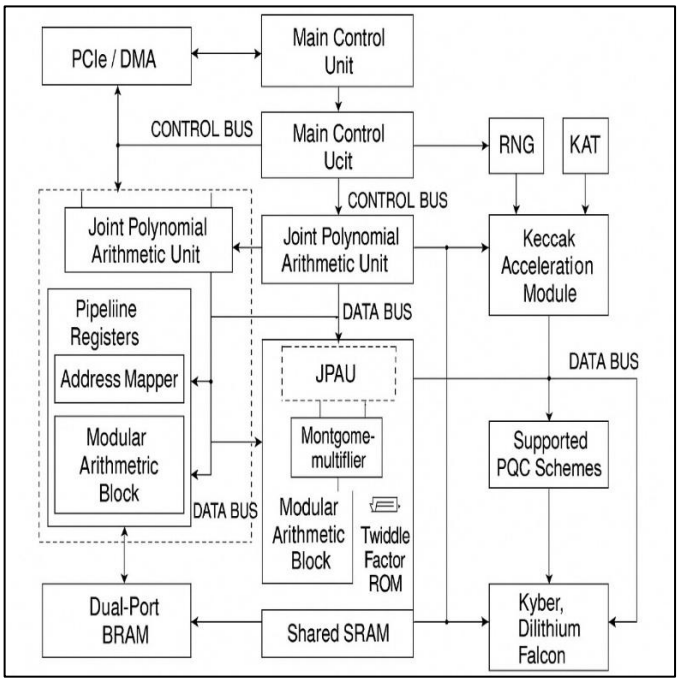

Fig 4.1 Hardware Architecture for Secure and Agile Post-Quantum Cryptographic Acceleration

4.5 *Compliance for hardware Vendors*

1. **KEY STORAGE –** FIPS Section 4.6.1 talks about zeroization . It shall occur in a sufficiently small time period so as to prevent the recovery of the sensitive data between the time of detection and the actual zeroization.
2. **CRYPTOGRAPHIC MODULE SECURITY – in appendix C FIPS**
   - **a.** Purpose
   - **b.** Diagram and each version should be mentioned
   - **c.** Security strength of module
   - **d.** How to implement function, exit and entry into the module.
   - **e.** Overall security and rules of operation
   - **f.** Cryptography module ports and interfaces to be listed (physical/logical). It should define the information that is passing over the four logical interface.
3. **AUTHENTICATION , ROLES AND SERVICES – appendix C FIPS**
   - **a.** Strength of authentication
   - **b.** Is there any capability to bypass authentication
   - **c.** Authentication status
4. **RNG RANDOM SOURCES -** There should be proper health test and validation of the RNG.
5. **SELF TEST/EVALUATION LOGIC –** Include KAT for the cryptography , to validate algorithm correctness at power up, on diagnostic on demand.
6. **LOGIC/ SCHEME SWITCHING –** The module should be able to switch based on the host requirement via a secure control like TaPaSCo that uses runtime logic scheduling based on flags.

4.6 *CARAF Framework*

Phase 1: Identification of Threats

Objective: To ascertain the specific threat vector (such as quantum computing, algorithmic vulnerabilities, or regulatory mandates) that necessitates the implementation of crypto agility.

Phase 2: Inventory of Assets

- **Scope -** proper scope should be defined
- **Sensitivity –** Higher expected risk must be identified and prioritized .
- **Cryptography** – assessing the security of algorithms used and appropriateness of the key length.
- **Management –** API tokens, certificates and frequency of updates.
- **Implementation** – Avoid hardcoded implementations or use of default credentials.
- **Ownership -** Complete information about the vendor and asset ownership should be recorded **.** Cryptography as a Service is an emerging trend in cloud computing. Even traditional computing widely uses open source crypto-libraries, such as OpenSSL. Documenting these upstream and downstream dependencies is critical as one risk of third party products

used by an organization is the lack of adequate and timely update.
- **Location –** it determines how secure and crypto agile the infrastructure is.

Phase 3: Risk Estimation

Objective: To quantify the risk associated with each asset or asset group employing a refined formula:

Risk = (Timeline to Exposure) × (Cost of Mitigation)
Timeline Breakdown (based on Mosca's Model):
- X: Asset lifespan or shelf-life
- Y: Duration required to mitigate or upgrade
- Z: Time until threat realization (for instance, quantum break)

Cost Components:
- Type of implementation (hardware versus software)
- Complexity of secrets rotation
- Effort required for cryptographic migration
- Vendor dependencies and contractual terms

Risk Levels:
- Risk exposure is scored on a scale of 1 to 4 (Low to Critical) for dimensions X, Y, and Z.
- Matrix analysis facilitates the classification of risk exposure for each asset.

Phase 4: Secure assets through risk management

- Secure assets by spending resources, when asset value is greater than the cost to secure.
- Accept the risk , when the estimated/ expected value of the risk is lower than organization's risk tolerance.
- Phase out impacted area, only when cost to secure is high along with the asset value being lower than that of expected risk.

Phase 5: Organisational Roadmap

- Incident Response Plan: Plans to address vulnerabilities in internally approved cyrpto-solutions.
- Third Party Risk Assessment: The vendors should be responsive to disclose the vulnerabilities discovered and prepare with immediate patch and update.
- Security Architecture Reviews: Assess the crypto agility of the proposed architecture and through review.
- Product Development: Development teams should be trained to select and understand the solutions with greater crypto agility. For example, when picking an open-source component, they should prioritize components with greater flexibility, better support, and faster patching history.
- Change Management Plan: Proper streamlined updates to be published and applied to cryptography solutions and supports infrastructure , e.g. key management systems, hardware security modules, etc

5. Software-Side PQC Implementation (avani)

5.1 Hybrid TLS Stack

In the process of preparing the existing financial infrastructure for the post-quantum era, a hybrid TLS stack is a robust transition strategy which maintains compatibility while introducing quantum-safe primitives. OpenSSL 3.5, extended with the Open Quantum Safe (OQS) project provider, enables the integration of classical key exchanges (e.g., X25519 Elliptic Curve Diffie-Hellman Key Exchange) with lattice-based key encapsulation mechanisms (KEM) like CRYSTALS-Kyber. This hybrid handshake structure ensures session confidentiality even if one cryptosystem (classical or PQ) is later compromised.

The implementation begins with building OpenSSL from source, and linking it with liboqs (libOpenQuantumSafe). Liboqs is a C library developed by OQS Project. It is a collection of PQC algorithms that are considered/selected by the National Institute of Standards and Technology (NIST), with a unified interface for using them in software. Once integrated, hybrid TLS configurations allow for Kyber768 + X25519 as the negotiated KEM suite. A Python-based Flask server can utilize this custom OpenSSL version to serve HTTPS endpoints supporting post-quantum TLS (PQTLS). The TLS 1.3 handshake sequence includes hybrid ClientHello and ServerHello extensions, with both public keys embedded, followed by derivation of a shared secret using both classical and PQ pathways. This ensures resilience against quantum attacks without breaking compatibility with legacy systems. Given that client-side support for PQTLS is still under development,

hybrid mode ensures backward compatibility with existing browsers and legacy applications while offering forward security.

The handshake requires both the client and server to support hybrid mode extensions, including the identifiers for supported PQ KEMs, as defined in the draft IETF proposal for hybrid key exchange. Notably, certificate parsing and KEM selection are handled using liboqs's TLS callback logic built into the OQS OpenSSL fork. The post-quantum KEM performs encapsulation on the client side and decapsulation on the server side. X25519-derived secrets and Kyber-derived secrets are then combined via HKDF-Extract to form the master secret. TLS handshakes incorporating Kyber-768 and classical X25519 have been shown to introduce an overhead of ~2–3 KB per handshake, but the impact on user experience is negligible for large payloads, as seen in the paper *TLS to PQTLS*.

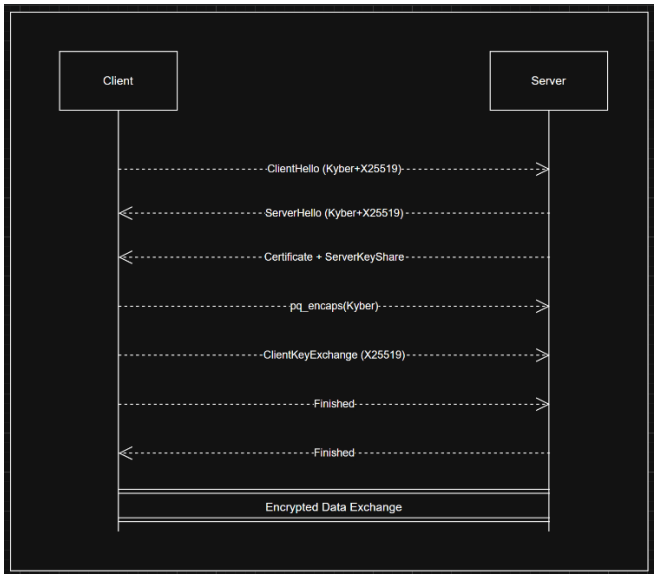

Figure 5.1 Flow Diagram of Hybrid TLSv1.3 Handshake

5.2 PQC JWT Signing

Secure token-based authentication, such as JWT (JSON Web Token), supports numerous stateless banking APIs. Replacing the signing mechanism of JWTs with post-quantum digital signature algorithms such as Crystals-DILITHIUM or FALCON provides resistance against future forgery attacks by quantum adversaries. Additionally, leveraging liboqs-python, bindings for the liboqs C library, allows developers to integrate NIST-approved signature algorithms into Python-based identity services.

For this implementation, 3 post quantum algorithms were used for experimentation: Dilithium3, Falcon512, and SPHINCS+. JWT claims were signed server-side using each scheme and validated via a custom verification endpoint. Among the tested algorithms, Dilithium3 offered balanced performance (~2.7KB signature, sub-10ms signing time), while Falcon512 produced compact signatures (~700B) but required floating-point arithmetic support, introducing complexity for some platforms. SPHINCS+, although stateless and hash-based, resulted in significantly larger JWTs (~8–30KB) and slower operations, making it suitable for offline or archival use cases rather than real-time token validation.

Performance benchmarks from the paper *Designing a Scalable and Area-Efficient Hardware Accelerator Supporting Multiple PQC Schemes* confirm these findings, where Dilithium3 consistently achieved optimal trade-offs for throughput and size across varying key sizes.

To extend JWT integration into a scalable system, support for algorithm identifiers can be included in the token header using JWS 'alg' fields, with parsing logic adapted to accept PQC values (e.g., Dilithium3, Falcon512, etc.). Keys can be generated via liboqs.KeyEncapsulation objects in Python and serialized in base64. Verification times and latency can be profiled using Python microbenchmarks. Such integrations can also benefit from dynamic algorithm selection, which is discussed further in 5.4.

5.3 Hybrid Certificates

To support compatibility with existing TLS clients while ensuring post-quantum authentication, hybrid certificates have emerged as a transitional mechanism. These certificates carry dual signatures: one using a classical algorithm (e.g., ECDSA) and the other using a PQC algorithm (e.g., Dilithium3). The certificate includes two public keys and associated algorithm identifiers encoded as ASN.1 OIDs, and clients validate both signatures when possible. This dual validation model provides continuity while ensuring future-proofing.

The implementation plan uses concatenated signatures within the same X.509 structure, aligning with hybrid cert models proposed by ISARA and

demonstrated in Cisco's PQC TLS stack. For clients that do not support PQ extensions, classical signatures ensure trust chain continuity. Meanwhile, PQ-aware clients can use the quantum-safe signature and validate both. Cross-certification can also be used for intermediate CAs, allowing a CA to be trusted under both classical and PQ roots. Although hybrid certificates are not yet standardized in major PKI toolchains, test implementations exist in forks of OpenSSL and BoringSSL, and experimental interop testing is underway.

Research outlined in *CARAF: Crypto Agility Risk Assessment Framework* discusses operational and encoding structures for hybrid certs and highlights the importance of consistent signature order and deterministic encoding for compatibility. Implementation guidance also emphasises on the need to limit cert chain length and optimize PQ signature sizes to mitigate handshake latency. Additional care should be taken to ensure compatibility with existing certificate validation libraries, especially when combining non-standard OIDs or key usage extensions.

5.4 Crypto Agility Integration

Cryptographic agility, defined as the ability to switch cryptographic primitives without redeploying application logic, is paramount in a post-quantum world. As new PQC standards emerge or new vulnerabilities are discovered, systems must be able to swap algorithms dynamically. Implementation agility allows for pluggable cryptographic modules, while configuration agility allows changing algorithm policies via external settings (JSON/YAML/ENV).

A recommended design pattern includes a central configuration file (e.g. ,cryptoconfig.yaml) that specifies algorithm use per operation (e.g., KEM: Kyber768, Digital Signature: Dilithium3). These configs are hot-reloadable via API endpoints (e.g., /config/algorithms) for dynamic control. In microservice-based infrastructures, each cryptographic operation, such as signing, verification, or key generation is encapsulated in its own stateless container, enabling independent updates and CI/CD integration.

To prevent downgrade attacks, fallback logic must be tightly governed by access policies and version constraints. The following YAML snippet serves as an example.

```yaml
signing_algorithm: Dilithium3
fallback: Falcon512
tls_kem: Kyber768
```

Figure 5.2: YAML snippet showing fallback logic

The use of containerized cryptographic services with proper interfaces (e.g., REST/gRPC) ensures that the rest of the infrastructure remains agnostic to algorithm updates. Internal PKI services can rotate keys and regenerate CSR using the updated algorithms. CI pipelines can validate algorithm-specific test vectors, and fallback procedures can load classical certs in case of PQC failure.

As outlined in the CARAF framework and PQC JWT case study in the paper *Introducing Cryptographic Agility in Mobile Banking*, cryptographic agility must support rollback mechanisms, modular APIs, and security policy orchestration at runtime. These strategies ensure systems remain resilient to future cryptanalytic advances while reducing update risk.

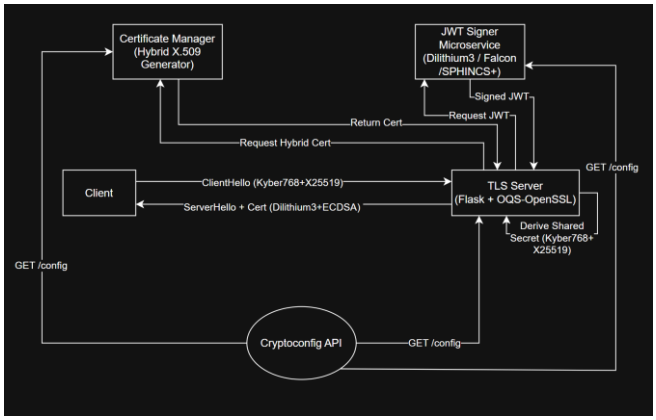

Figure 5.3: Overall Flow Diagram of the Software of the Proposed Implementation

6.1 Microservices vs. Monoliths: Encryption in Fintech

Robust encryption is necessary for modern fintech/mobile banking systems to secure sensitive data (accounts, payments, PII) under stringent rules (PCI-DSS, PSD2, etc.) both in transit and during processing. The database, business logic, and user

interface all reside in a single unit in a monolithic system. Calls made within are "trusted" by default; therefore, only the perimeter is frequently encrypted. A microservices architecture, on the other hand, divides the application into numerous tiny, independent services. This naturally leads to a "zero-trust" strategy since all service-to-service communications must be encrypted and authenticated over a network. Therefore, compared to monoliths, microservices provide more robust and adaptable encryption capabilities. Specifically, microservices isolate the data and credentials of each service while enabling per-service certificates/keys, automated mutual TLS (mTLS), and dedicated tokenization. Finer-grained security is the end result.

- **Smaller attack surfaces & isolation:** Because each microservice has a tiny scope and few duties, it has a considerably smaller attack surface. A single vulnerability in a monolith can reveal the entire codebase, yet a compromised microservice can be isolated or terminated without erasing the entire system. One fintech migration, for instance, pointed out that by keeping modules "separated", data in other services is protected in the event of a breakdown in one service. Teams may also encrypt and protect each service separately thanks to isolation; if the keys to one service is compromised, just that service's data—not all data is in danger.
- **Independent encryption policies:** Each service can implement its own crypto methods and encryption/authentication rules thanks to microservices. This implies that updating or rotating the TLS/crypto libraries for a single service doesn't need redeploying the entire application. In contrast, it is frequently necessary to rebuild the entire application when changing encryption in a monolith. Microservices can individually store their own encryption keys (often through a secret manager) because they typically operate in containers or on different hosts. Teams usually set up automated key rotation or revocation and provision each service with a TLS certificate or key. To put it briefly, at the service level, encryption becomes automated and flexible, while at the monolith, major modifications would be required.
- **Zero-trust by design:** Zero-trust principles are naturally implemented in microservices architectures. All network calls need to be permitted and authenticated. For instance, rather than relying on internal calls, NIST and others advise mutual authentication between microservices. A monolith, on the other hand, does not authenticate any of its internal modules. All inter-service communication is encrypted end-to-end when mTLS is enforced between microservices, and both parties authenticate one another. This significantly raises the bar for lateral or man-in-the-middle attacks.
- **Easier auditing and compliance:** Microservices make it simpler to track down which service accessed which data because they log every request and use different credentials. It is possible to log and audit encryption keys for each service. When services are individually secured, it is simpler to comply with compliance laws (such as PSD2's requirement for robust consumer authentication).

6.2 Mutual TLS, Tokenization, and Real-Time Encryption

Mutual TLS (mTLS): Two-way TLS between services can be automatically enabled via a service mesh (like Istio or Linkerd). Every service using mTLS has a public/private key pair, and before exchanging data, the client and server authenticate themselves. As a result, all service-to-service communication is guaranteed confidentiality and integrity. For instance, in order to comply with PSD2, Finleap Connect, an open-banking platform, deployed mTLS on each microservice in its cluster. In less than a week, they were able to protect more than fifty services in production using Linkerd. Since internal calls in a monolith are unchecked and TLS is only utilized at the front end, the idea of mTLS is irrelevant. mTLS is a common technology used in microservices to enforce zero-trust encryption across the wire.

Tokenization: Sensitive field processing can be delegated by microservices to a specific tokenization service. Tokenization substitutes irreversible tokens for raw data, such as account numbers and card PANs. After that, other services merely use tokens rather than actual data. This implies that for fintech apps that

handle payments, all other services are protected, and only the tokenization service ever sees the plaintext. For PCI-DSS compliance, this is standard procedure (e.g., credit card tokens). In contrast to microservices, which may centralize tokenization and remove plaintext outside of a private vault, monoliths frequently employ a single database and may store sensitive settings directly.

Real-time ("on-the-fly") encryption: A lot of microservices encrypt data while it's being processed using hardware enclaves or encryption-as-a-service. The transit engine of HashiCorp Vault, for example, has the ability to encrypt data on demand prior to it leaving a service and decode it upon arrival. Without managing keys, a microservice encrypts and decrypts data by only using Vault. This method separates application code from crypto. Retrofitting on-the-fly encryption was challenging since monoliths, particularly older ones, seldom had built-in support for such runtime encryption. Runtime data encryption and format-preserving encryption may be gradually introduced to services in microservices without compromising other services.

6.3 Secure Communication via Isolation

Service Mesh / API Gateway: Between services are tools like Gloo Mesh, Linkerd, and Istio. They enforce policies (ACLs, quotas) and transparently encrypt and decode all traffic (TLS). It is not necessary for developers to implement crypto themselves. This really implies that in a microservices cluster, inter-service calls are always encrypted by default.

Network Segmentation: Microservices can operate in different containers, virtual machines, or clusters. Only certain services can communicate with one another if you use firewall rules or network restrictions. To reach downstream services using mTLS, for instance, a payment service can only accept requests from an authorized API gateway. This "defense in depth" strategy, which consists of encrypted links and logical isolation, stops an attacker from moving freely.

Least Privilege & IAM: Strong identification systems (OAuth/JWT, SPIFFE IDs) are used by microservices for each service. A token or certificate is carried by each service call. A monolith, on the other hand, usually does not enforce service-to-service identification; any code is regarded as trustworthy once you are within the application.

Logging and Auditing: Each microservice barrier functions as a security checkpoint when various isolation techniques are combined. Every piece of data that crosses a barrier is authenticated and encrypted. These internal checks are absent from monolithic programs, which only use perimeter protections, which may be single points of failure.

6.4 Fintech/Banking Case Studies

Finleap Connect (2022)

In order to comply with PSD2, Finleap, a European open-banking platform that serves banking and accounting firms, redesigned its system using microservices. They used Linkerd to "mTLSed" all 50+ microservices that were spread over 10 clusters over the course of five months. They said that setting up mTLS "was fairly easy" and that it just took an hour to deploy throughout production, allowing for encryption and mutual authentication for each API request between services. Strict legal criteria for end-to-end encryption of private payment information were met by this.

6.5 NORD/LB (2019–2020)

A hybrid cloud microservices infrastructure is used by this German bank. They used HashiCorp Vault to automate data encryption while it's in transit and at rest and to centralize key management. Data is dynamically encrypted by Vault's Transit engine when services exchange messages. Prior to Vault, key rotation required four days of human labor; now, it takes less than five minutes. Regulators required a digital audit record of all crypto operations, which Vault provided. By ensuring that every microservice utilizes its own dynamically provided keys, NORD/LB significantly increases the agility of encryption.

Large Banks (e.g. Citibank, Capital One, etc.)

Microservices are used by several international banks for their key systems. For instance, service meshes and microservices are used by Capital One's cloud platforms to isolate data and implement TLS everywhere. The move to microservices in banking is

frequently motivated by encryption and compliance requirements, even if the specifics aren't made public.

6.6 Monolithic Challenges – and Microservices Solutions

Uniform Trust Domain

Internal parts of a monolith inherently trust one another. Usually, authentication and encryption don't need to be done "inside" the program. However, this implies that all data is exposed in the event of an app (or database) breach. Microservices, on the other hand, require authentication at each step. In contrast to microservices, where "you must make sure [each] request is authenticated," a monolith "has all the resources it needs…no need for authentication within," according to one engineer.

Static Keys & Secrets

Monoliths frequently employ static keystores or hard-code secrets. Older financial apps sometimes lack encryption entirely since they were developed before contemporary crypto libraries. The keys are usually static, even with encryption applied (e.g. a single key in a config file). Redeploying the entire application is necessary in order to rotate those keys. In contrast, microservices make use of secrets managers, often known as HSMs. At launch, each service retrieves ephemeral keys, which may be cycled individually without causing any downtime (as in the case of the NORD/LB Vault).

Single DB/Storage Encryption

One or a small number of centralized databases are often found in a monolith.
That whole database is coarse-grainedly encrypted. Large volumes of data might be decrypted by an attacker if a key is compromised. In contrast, microservices frequently make use of distinct data stores for each service. A unique key can be used to encrypt each storage. Only one service's data is impacted by a compromised key; the system as a whole is unaffected.

Slower Updates

Security updates for monoliths are more difficult. The entire application must frequently be rebuilt and retested in order to fix or update the crypto library. With microservices, you can implement encryption updates incrementally, changing the runtime or library of a single service at a time. This implies that new cipher suites or enhanced protocols (like TLS1.3) can be progressively implemented.

No Fine-Grained Control

Micro-granular controls, such as sidecars or API gateways, are absent from monoliths. For certain sections of the program, logging, rate-limiting, and mutual auth are difficult to implement. With microservices, you may add mTLS or token validation right away without changing business code by dropping a new service mesh proxy in front of any service.

Conclusion

In conclusion, our microservices-based framework fundamentally transforms encryption control and algorithm agility by effectively decomposing cryptographic functions into independent services. Each microservice strictly enforces its crypto policy, enabling real-time upgrades and swaps without the need to redeploy the entire application. This architecture isolates faults: if one service encounters an issue, it does not compromise the entire system. Our centralized software-defined crypto layer expertly manages governance, facilitating seamless transitions to new NIST-approved post-quantum algorithms without any code rebuilds. With FIPS 140-2 Level 3 hardware security modules and secrets managers in place, we provide a robust root of trust that guarantees compliance as PQC algorithms evolve.

This structure delivers a resilient, scalable platform that confidently meets the rigorous availability, security, and audit requirements of regulated mobile banking infrastructures. Moreover, financial firms are mandated to tackle quantum threats head-on, making crypto-agility an essential strategy. Thus, this work stands as a compelling proof-of-concept for a quantum-safe fintech platform featuring a secure, adaptable microservice-driven crypto architecture poised to meet future challenges with ease and minimal disruption.

References/ Citations:

[1] Cloud Native Computing Foundation, “Case Study: finleap connect,” CNCF.io, 2022. Available: https://www.cncf.io/case-studies/finleap-connect/

[2] Frontegg, “Authentication in Microservices: Best Practices,” Frontegg Blog, 2023. Available: https://frontegg.com/blog/authentication-in-microservices

[3] HashiCorp, “Case Study: Nord/LB Modernizes IT with Vault,” Hashicorp.com, 2023. Available: https://www.hashicorp.com/case-studies/nord-lb

[4] HQSoftware, “Banking & Fintech Microservices Architecture: Benefits,” HQSoftware, 2023. Available: https://hqsoftwarelab.com/blog/banking-fintech-microservices-architecture-benefits/

[5] A. K. Jain and M. Mishra, “Introducing Cryptographic Agility in Mobile Banking,” in Proc. 2023 8th Int. Conf. on Computing, Communication and Security (ICCCS), 2023, pp. 1–6. doi: 10.1109/ICCCS58268.2023.10417052

[6] M. Jain, R. Kumar, P. Ghosal, and R. Misra, “Design of Efficient Unified NTT Architecture for CRYSTALS-Kyber and CRYSTALS-Dilithium,” Electronics, vol. 13, no. 17, p. 3360, 2024.

[7] N. Karia and H. Lakhani, “A Unified and Scalable NTT Multiplier Architecture for Post-Quantum Cryptographic Schemes,” arXiv preprint arXiv:2311.04581, Nov. 2023. Available: https://arxiv.org/abs/2311.04581

[8] J. Lee, S. Kim, and H. Kim, “TLS → Post-Quantum TLS: Inspecting the TLS Landscape for PQC Adoption on Android,” in Proc. 2023 IEEE European Symp. on Security and Privacy (EuroS&P), 2023, pp. 501–515. doi: 10.1109/EuroSP56284.2023.00035

[9] L. Maier, P. M. Alvarez, C. Guenther, A. Schaller, and A. Wachter-Zeh, “PQC-HA: A Framework for Prototyping and In-Hardware Evaluation of Post-Quantum Cryptography Hardware Accelerators,” arXiv preprint arXiv:2308.06621, Aug. 2023. Available: https://arxiv.org/abs/2308.06621

[10] National Institute of Standards and Technology, “Recommendation for the Entropy Sources Used for Random Bit Generation,” NIST Special Publication 800-90B, Jan. 2018.

[11] National Institute of Standards and Technology, “Security Requirements for Cryptographic Modules,” FIPS PUB 140-3, Final Draft, Sept. 2007.

[12] OWASP, “Microservices Security Cheat Sheet,” OWASP Cheat Sheet Series, 2022. Available: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html

[13] Qwiet AI, “Microservices Security: Isolating and Protecting Your Service,” Qwiet.ai, 2023. Available: https://qwiet.ai/microservices-security-isolating-and-protecting-your-service/

[14] Spartan Solutions, “Encryption vs Tokenization: Which is Better for Fintech Security?” Spartan Blog, 2023. Available: https://www.spartansolutions.co/blog/encryption-vs-tokenization-which-is-better-for-fintech-security

[15] Traceable AI, “6 New Requirements for Securing Microservices Versus Monolithic Applications,” Traceable.ai, Sep. 2023. Available: https://www.traceable.ai/blog-post/6-new-requirements-for-securing-microservices-versus-monolithic-applications

[16] Townsend Security, “Why Encryption Is Critical to Fintech,” TownsendSecurity.com, 2023. Available: https://info.townsendsecurity.com/why-encryption-is-critical-to-fintech

[17] P. Upadhyay, B. Rawal, and N. Pradhan, “A Framework for Migrating to Post-Quantum Cryptography: Security Dependency Analysis and Case Studies,” in Proc. 2023 IEEE Int. Conf. on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), 2023, pp. 7–15. doi: 10.1109/TPS-ISA58304.2023.10417052

[18] M. Vasserman, R. Shapiro, D. Devendran, M. Kaczmarek, and B. Fisch, “CARAF: A Framework for Crypto Agility Risk Assessment,” Journal of Cybersecurity, vol. 7, no. 1, 2021.

[19] Invicti Security, “Monolithic vs Microservices Architecture: Which Is Better for Security?” Invicti.com, Sep. 2023. Available: https://www.invicti.com/blog/web-security/monolithic-vs-microservices-architecture-which-is-better-for-security/