암호민첩성 구현을 위한 미들웨어 프레임워크 설계

2025.10.24

성신여자대학교 융합보안공학과
김성민 (sm.kim@sungshin.a.ckr)

Contents

배경: PQC 전환 동향

[NIST PQC 알고리즘 표준화 현황]

- **2022**년) 최종 표준 알고리즘 4종 발표
  - ✓ Public-Key Encryption(PKE)/Key Encapsulation Mechanism(KEM) 부문 **1** 종
  - ✓ Digital Signature 부문 **3**종
- **2025**년)
  - ✓ 추가 KEM 알고리즘 **1**종 선정(HQC)

PKE/KEM	DSA
• ML-KEM • HQC	• ML-DSA • SLH-DSA • FALCON

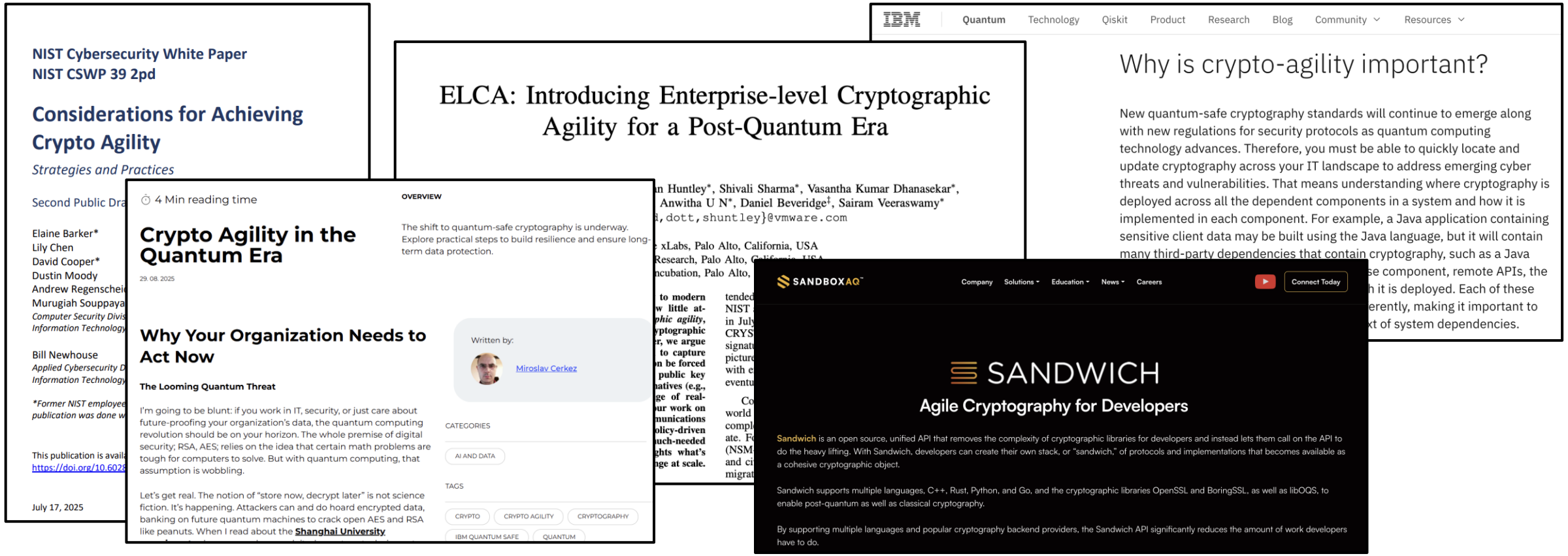
[KpqC 알고리즘 표준화 현황]

- **2025**년)
  - ✓ PKE/KEM 부문 **2**종
  - ✓ Digital Signature 부문 **2**종

PKE/KEM	DSA
• NTRU+ • SMAUG-T	• AIMer • HAETAE

배경: 암호 민첩성(Crypto Agility)의 필요성

- 기존 공개키 암호 기반 시스템 → 신규 암호 알고리즘으로 **효율적인 적용·전환**(≈**Crypto Agility**)에 대한 중요성 확대
  - KEM: ECDH/DH (pre-quantum) → ML-KEM (post-quantum)
  - DSA: ECDSA/RSA (pre-quantum) → ML-DSA (post-quantum)
- But, **암호 민첩성**을 위한 전환 관점에서의 연구는 초동 단계



암호 민첩성(Crypto Agility) 정의 및 주요 속성

[BSI Definition] [BSI]

- A cryptosystem is considered crypto-agile if its components, for example cryptographic algorithms, key lengths, key generation schemes or technical implementation, can be replaced by other components without having to make significant changes to the rest of the overall system.

[NIST Definition] [CSWP39]

- The ability 1) for machines to select their security algorithms in real time and based on their combined security functions; 2) to add new cryptographic features or algorithms to existing hardware or software, resulting in new, stronger security features; 3) to easily retire cryptographic systems that have become either vulnerable or obsolete.

BSI VS. NIST

Protocol(Algorithm)-centric agility

- ✓ 기술/설계 관점에서 알고리즘 교체 용이성에 방점
- ✓ Functional consistency vs. Interoperability

Developer(Operator)-centric agility

- ✓ 운영/배포 관점에서 상호운용성에 방점
- ✓ Functional consistency vs. Interoperability

*** 현재 암호 민첩성 개념의 정의는 암호 민첩성 시스템을 구현함에 있어 notion 정의의 모호성 및 design space가 존재**

[CSWP39] Barker E, Chen L, Cooper D, Moody D, Regenscheid A, Souppaya M, Newhouse B, Housley R, Turner S, Barker W, Scarfone K (2025) Considerations for Achieving Crypto Agility: Strategies and Practices. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Cybersecurity White Paper (CSWP) NIST CSWP 39 2pd. https://doi.org/10.6028/NIST.CSWP.39.2pd
[BSI] Bundesamt für Sicherheit in der Informationstechnik (BSI), Krypto-Agilität – Empfehlungen zur Umsetzung, BSI TR-02102, 2021.(In German)

암호 민첩성(Crypto Agility) 정의 및 주요 속성

[Primary Goal]

- Security, Policy-awareness: 보안성을 보장해야 함
- Heterogeneity, interoperability: 이종의 암호 primitive들을 지원하고, 상호운용 가능해야 함

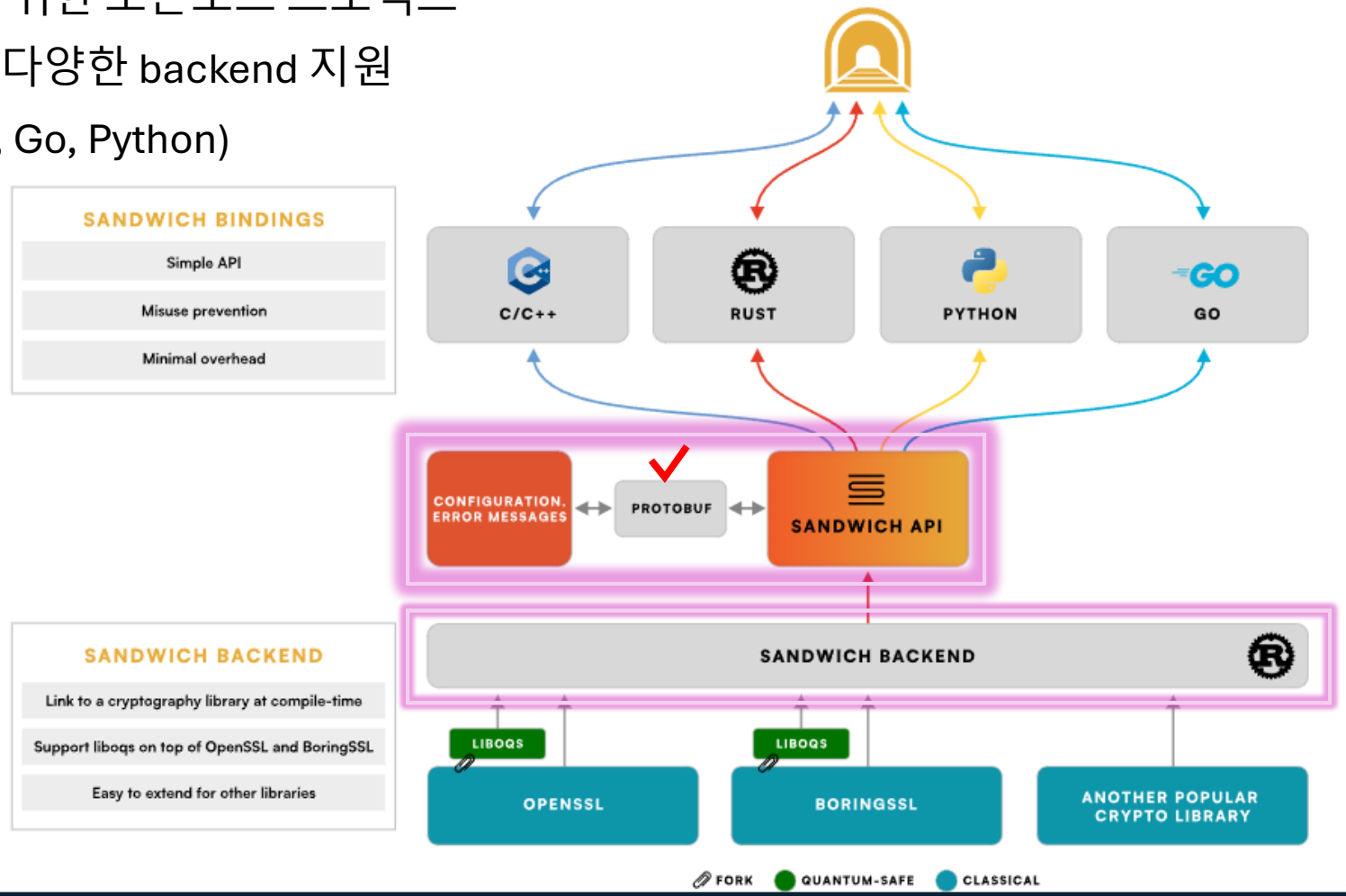
But also need to provide

- Backward-compatibility: 레거시 프로토콜/알고리즘 기반의 라이브러리/에플리케이션에 수정 없이 적용 가능해야 함.
- Effectiveness, performance: 성능 효율성을 보장해야 함
- Extendibility, flexibility, upgradeability: 암호 프로토콜/알고리즘을 추가하거나 제거, 확장 가능해야 함.

Sandwich API & Backend

[Potential baseline for agility-aware API design: SANDBOXAQ's Sandwich API]

- 동적인 Crypto Agility를 구현하기 위한 오픈소스 프로젝트
- LIBOQS, OpenSSL, BoringSSL 등 다양한 backend 지원
- 다양한 언어 바인딩(C/C++, Rust, Go, Python)



출처: https://github.com/skydoves/sandwich

Sandwich API & Backend

[Potential baseline for agility-aware API design: SANDBOXAQ's Sandwich API]

Crypto-agility 구현을 위한 메타 라이브러리로 활용하기에 적합

- "Meta-library": 여러 암호 라이브러리(예: OpenSSL, BoringSSL, libOQS 등)를 통합하는 상위 API 제공
- 이때, 각 backend는 pluggable한 구조를 가짐 (compile/runtime selection이 모두 가능)

암호 알고리즘과 프로토콜을 추상화하여, 실제 구현(backends)을 손쉽게 교체 가능하도록 설계

- 개발자가 "protocol + implementation" 조합으로 Sandwich 객체 생성을 통해 구성 변경 가능
- Unified API로 손쉽게 암호구성 모듈을 조립하고 변경 가능
- 코드 변경 없이 구성파일(혹은 파라미터)만으로 backend, 알고리즘, 키 교체 가능

출처: https://github.com/skydoves/sandwich

Sandwich API & Backend 분석

[세부 API 설계: Tunnel abstraction]

- `Tunnel`이라는 추상화 객체를 도입, TLS 등 암호화 채널 구성을 관리하는 데 활용
- `Tunnel` 객체와 High-level wrapper API 형태로 작업들을 처리하도록 구현 가능
  - 알고리즘, 백엔드 선택, 프로토콜 등을 파라미터로 전달하여 코드 변경 없이 교체 가능

```
from sandwich import Tunnel
tunnel = Tunnel(protocol="tls13", backend="openssl", ...)
tunnel.handshake(...)
tunnel.send(...)
tunnel.recv(...)
```

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "openssl", "liboqs"
  string cipher_suite = 3;       // "TLS_AES_256_GCM_SHA384"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
```

- Crypto Agility 개념에서 'extensibility/Upgradability' & 'Heterogeneity/interoperability'에 초점을 맞춤
  ☑ Centralized policy control을 위한 basis가 될 수 있는 abstraction(`TunnelConfig`) 제공

Sandwich API & Backend 분석

[세부 API 설계: Tunnel abstraction]

- `Tunnel`이라는 추상화 객체를 도입, TLS 등 암호화 채널 구성을 관리하는 데 활용
- `Tunnel` 객체와 High-level wrapper API 형태로 작업들을 처리하도록 구현 가능
  - 알고리즘, 백엔드 선택, 프로토콜 등을 파라미터로 전달하여 코드 변경 없이 교체 가능

```
from sandwich import Tunnel
tunnel = Tunnel(protocol="tls13", backend="openssl", ...)
tunnel.handshake(...)
tunnel.send(...)
tunnel.recv(...)
```

High-level wrapper APIs

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "openssl", "liboqs"
  string cipher_suite = 3;       // "TLS_AES_256_GCM_SHA384"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
```

- Crypto Agility 개념에서 'extensibility/Upgradability' & 'Heterogeneity/interoperability'에 초점을 맞춤
  - ☑ Centralized policy control을 위한 basis가 될 수 있는 abstraction(`TunnelConfig`) 제공

Sandwich API & Backend 분석

[세부 API 설계: Tunnel abstraction]

- `Tunnel`이라는 추상화 객체를 도입, TLS 등 암호화 채널 구성을 관리하는 데 활용
- `Tunnel` 객체와 High-level wrapper API 형태로 작업들을 처리하도록 구현 가능
  - 알고리즘, 백엔드 선택, 프로토콜 등을 파라미터로 전달하여 코드 변경 없이 교체 가능

```
from sandwich import Tunnel
tunnel = Tunnel(protocol="tls13", backend="openssl", ...)
tunnel.handshake(...)
tunnel.send(...)
tunnel.recv(...)
```

High-level wrapper APIs

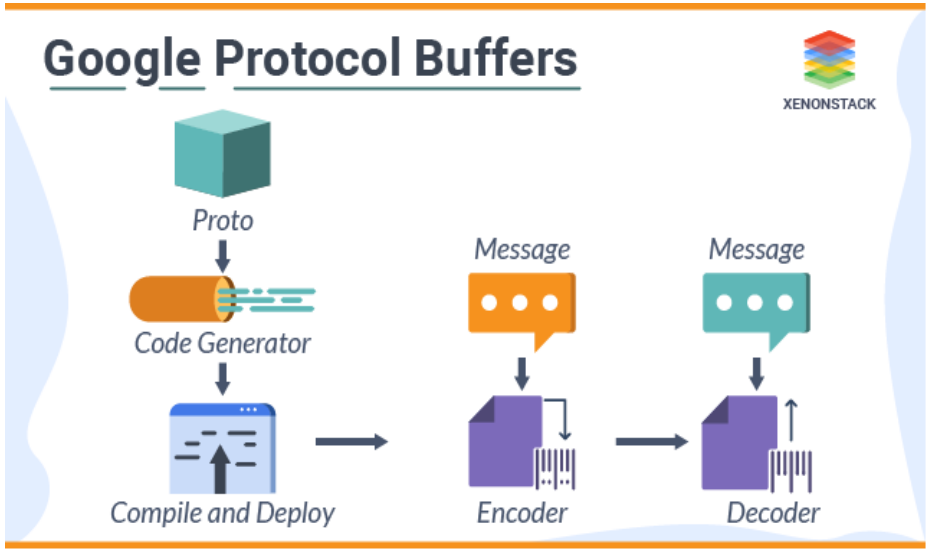
```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "openssl", "liboqs"
  string cipher_suite = 3;       // "TLS_AES_256_GCM_SHA384"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
```

- Crypto Agility 개념에서 'extensibility/Upgradability' & 'Heterogeneity/interoperability'에 초점을 맞춤
  - ☑ Centralized policy control을 위한 basis가 될 수 있는 abstraction(`TunnelConfig`) 제공

Sandwich API & Backend 분석

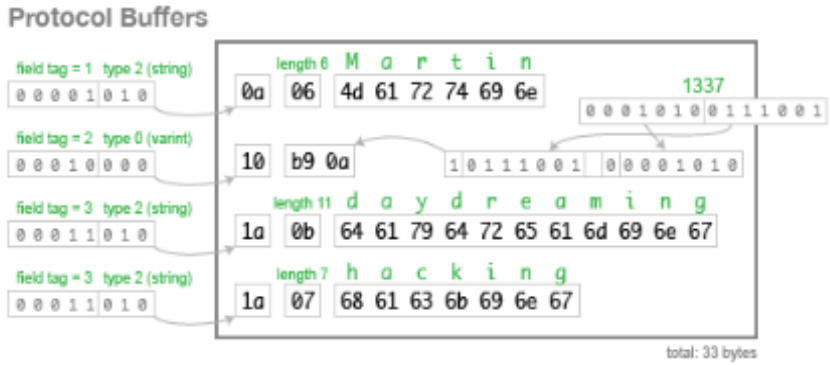
[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **Protocol Buffer**
  - 구글이 개발한 언어/플랫폼 독립적 바이너리 직렬화 포맷
  - 데이터 구조를 .proto 파일에 정의, 다양한 언어로 자동 코드 생성
  - 경량화/고성능 데이터 파싱을 보장, 명확한 타입 schem를 통해 정적 타입 체크 및 자동화 측면에서 강점을 가짐
  - 호환성: 버전 관리 및 확장 용이(필드 번호, optional 등으로 하위/상위 호환)




```
{
    "userName": "Martin",
    "favouriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

```
message Person {
    required string user_name        = 1;
    optional int64  favourite_number = 2;
    repeated string interests        = 3;
}
```

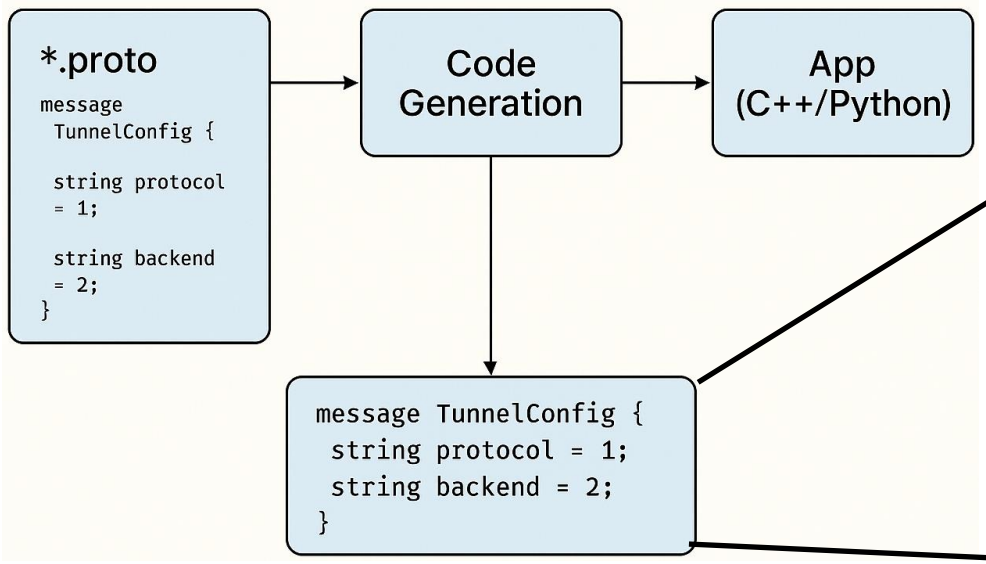


Sandwich API & Backend 분석

[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **Sandwich에서의 Protobuf 설계 구조**
  - Sandwich API는 주요 설정(config, handshake, tunnel 등)을 protobuf로 표준화
  - 언어별 바인딩 및 런타임 구성, 정책 변경을 protobuf 메시지로 일관성 있게 관리

- **Sandwich에서의 TunnelConfig 메시지 정의 (TunnelConfig를 .proto 파일에 작성)**



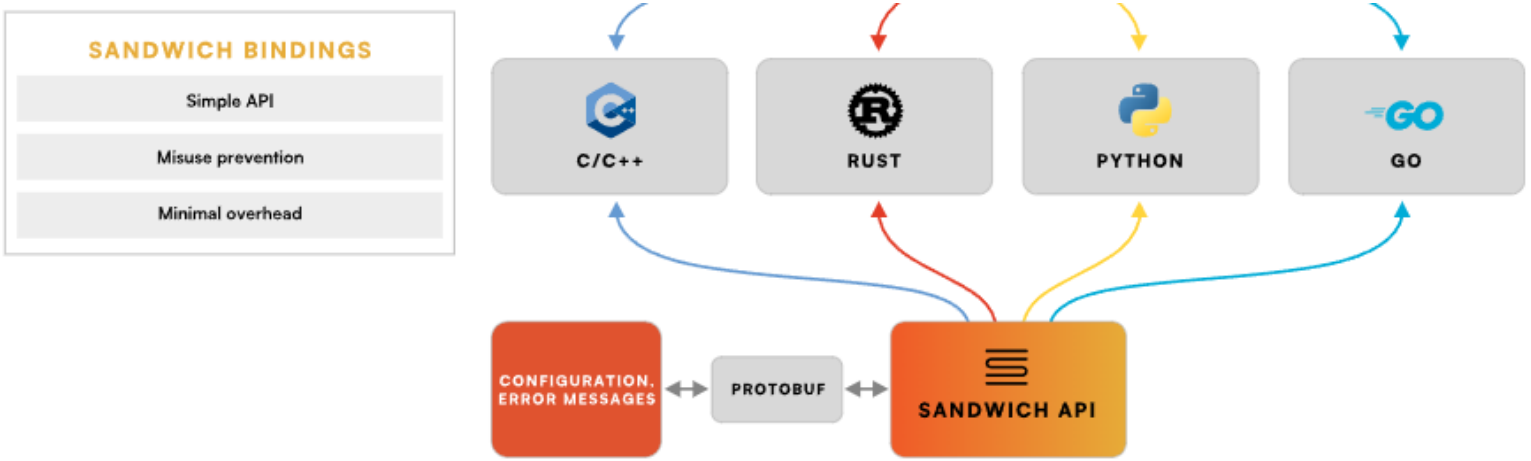

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "openssl", "liboqs"
  string cipher_suite = 3;       // "TLS_AES_256_GCM_SHA384"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
}
```

Sandwich API & Backend 분석

[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **Protobuf 활용의 장점**
  - 동일 TunnelConfig 메시지를 C++, Python, Rust, Go 등 다양한 언어에서 활용

    → protoc 코드생성 도구로 언어별 라이브러리 생성을 통해 로직 코드와 설정을 분리 **(Modular)**
  - 중앙 설정/정책 파일의 일관된 분배와 관리 용이 **(Centralized policy control)**



Sandwich API & Backend 분석

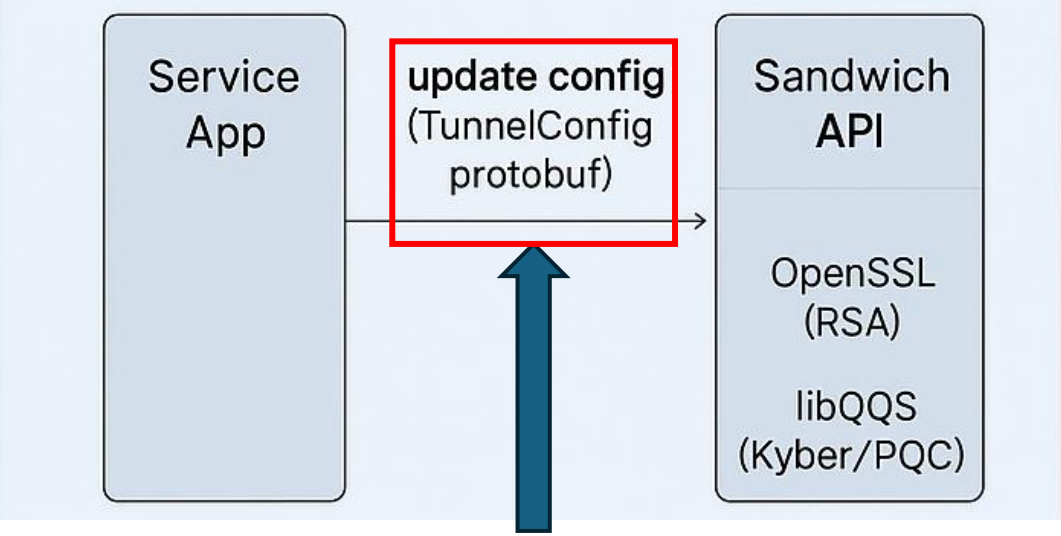
[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **Protobuf 활용의 장점**
  - 새로운 암호 알고리즘 추가 시, proto 메시지에 필드만 확장하면 하위 호환성 유지

    → 기존 메시지 필드 제거는 금지, 새 필드는 번호만 유니크하게 추가 **(Backward compatible & extensibility)**

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;          // "tls13"
  string backend = 2;           // "openssl", "liboqs"
  string cipher_suite = 3;      // "TLS_AES_256_GCM_SHA384"
  string kem = 4;               // "kyber768" (PQ)
  string signature_scheme = 5;  // "dilithium2"
```

// 신규 확장 시 TunnelConfig 필드 추가 (기존 시스템 영향 X)
string extra_pqc_param = 6;



Config message 포맷과 인코딩/디코딩 되는
protobuf message만 달라지므로 로직 변화 필요 X

Sandwich API & Backend 분석

[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **언어 별 호출 예시**
  - protobuf 기반 config만 전달하면 언어별 사용법이 사실상 동일 (cfg setting + generic API call)

Python

```python
from sandwich import Tunnel

cfg = Tunnel(protocol="tls13", backend="liboqs", kem="kyber768")
cfg.handshake(...)
cfg.send(b"hello")
data = cfg.recv()
```

Rust

```rust
let mut tunnel = Tunnel::new("tls13", "liboqs").with_kem("kyber768");
tunnel.handshake()?;
tunnel.send(b"hello")?;
let data = tunnel.recv()?;
```

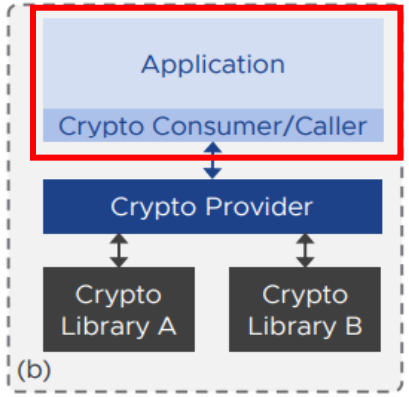
C++

```cpp
TunnelConfig cfg;
cfg.set_protocol("tls13");
cfg.set_backend("liboqs");
cfg.set_kem("kyber768");

Tunnel tunnel(cfg);
tunnel.handshake();
tunnel.send("hello");
auto data = tunnel.recv();
```

Sandwich API & Backend 분석

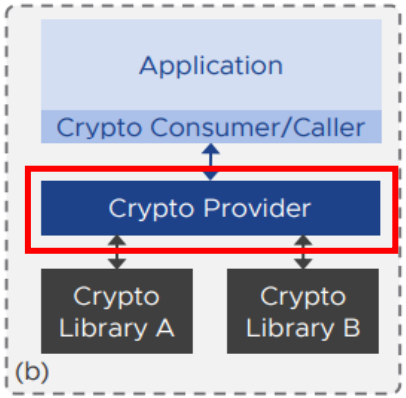
[세부 API 설계: Protocol Buffer (Protobuf) 기반 설계]

- **Crypto Agility를 위한 Protobuf 기반 구성 변경**
  - Server/Client side: Tunnel config update를 위한 YAML 파일 작성
  - Crypto provider (middleware): YAML 파일을 input으로 받아, protobuf message 변환 후 Sandwich API에 전달




```yaml
# tunnel_config.yaml
protocol: tls13
backend: liboqs
kem: kyber768
signature_scheme: dilithium2
```

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "liboqs"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
```



- **Is it fully backward compatible with legacy applications?**
  - **No.** Server/Client의 경우, Sandwich API를 쓰도록 application re-implementation 및 build가 필요
  - 다만, 한번 포팅 이후 crypto library 교체는 config 및 message 업데이트 만으로 재컴파일 없이 런타임에 가능

Sandwich API & Backend 분석

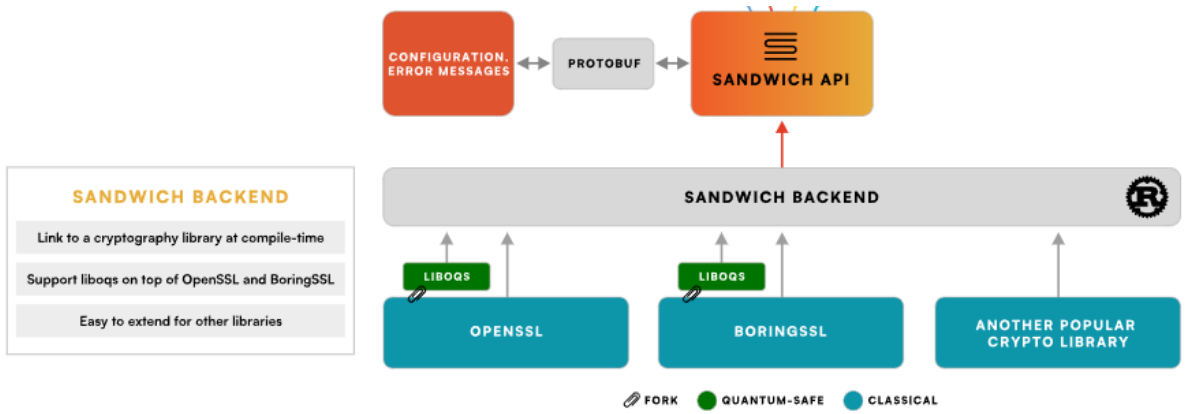
[Crypto-agility 구현 예시: Hands-on examples in the public repo]

- **기존 구성 (e.g., OpenSSL + RSA)**
  - TunnelConfig.protocol = "tls13
  - "TunnelConfig.backend = "openssl"
  - TunnelConfig.cipher_suite = "TLS_AES_256_GCM_SHA384"
  - TunnelConfig.signature_scheme = "rsa_pss_rsae_sha256"

- **PQC 대응 (libOQS + Kyber/Dilithium)**
  - TunnelConfig.protocol = "tls13"
  - TunnelConfig.backend = "liboqs"
  - TunnelConfig.kem = "kyber768"
  - TunnelConfig.signature_scheme = "dilithium2"

<Crypto 모듈 교체를 위한 procedure>

1. 사용자가 TunnelConfig configuration (YAML 파일) 작성
2. Sandwich client/server app.이 TunnelConfig protobuf 메세지로 변환 및 Sandwich API로 전송
3. Sandwich API가 Tunnel object에 따라 대응되는 adapter 모듈로 교체되도록 backend에 요청



Sandwich Summary

[Sandwich API & Backend 구현 현황]

- **아직 early stage의 pilot project 수준 (현재 v0.x)**
  - 현재 TLS Server/Client PoC implementation에서는 Subject Alternative Name (SAN) 인증서 검증만 구현
    ※ X.509 확장, TLS/SSL 인증서에서 여러 도메인이나 IP 주소를 하나의 인증서에 포함시킬 수 있게 해주는 기능
  - 키관리(KMS), 인증, VPN 등 추상화에 대해서는 현재 지원하지 않는 상황이 추후 확장 예정

- **Crypto-agility 개념에서 extensibility/Upgradability & Heterogeneity/interoperability에 초점을 맞춤**
  - Centralized policy control을 위한 basis가 될 수 있는 abstraction(TunnelConfig)은 제공
  - Consistent한 자동 암호 scheme/protocol/algorithm 교체에 대한 abstraction은 제공하지 않음
  - Resource constraint이나 SLO (Service-level Objectives) 등 추가 시스템 요구사항에 대한 고려는 X

Sandwich Summary

[세부 API 설계: Tunnel abstraction]

- `Tunnel`이라는 추상화 객체를 도입, TLS 등 암호화 채널 구성을 관리하는 데 활용
- `Tunnel` 객체와 High-level wrapper API 형태로 작업들을 처리하도록 구현 가능
  - 알고리즘, 백엔드 선택, 프로토콜 등을 파라미터로 전달하여 코드 변경 없이 교체 가능

```
from sandwich import Tunnel
tunnel = Tunnel(protocol="tls13", backend="openssl", ...)
tunnel.handshake(...)
tunnel.send(...)
tunnel.recv(...)
```

High-level wrapper APIs

```
syntax = "proto3";

message TunnelConfig {
  string protocol = 1;           // "tls13"
  string backend = 2;            // "openssl", "liboqs"
  string cipher_suite = 3;       // "TLS_AES_256_GCM_SHA384"
  string kem = 4;                // "kyber768" (PQ)
  string signature_scheme = 5;   // "dilithium2"
```

- Crypto Agility 개념에서 'extensibility/Upgradability' & 'Heterogeneity/interoperability'에 초점을 맞춤
  - ☑ Centralized policy control을 위한 basis가 될 수 있는 abstraction(`TunnelConfig`) 제공
  - ☐ **자동화된 암호 scheme/protocol/algorithm 교체에 대한 abstraction 제공**
  - ☐ **Resource constraint이나 Security level 등 교체 알고리즘 선택에 대한 requirement 고려**

Contribution

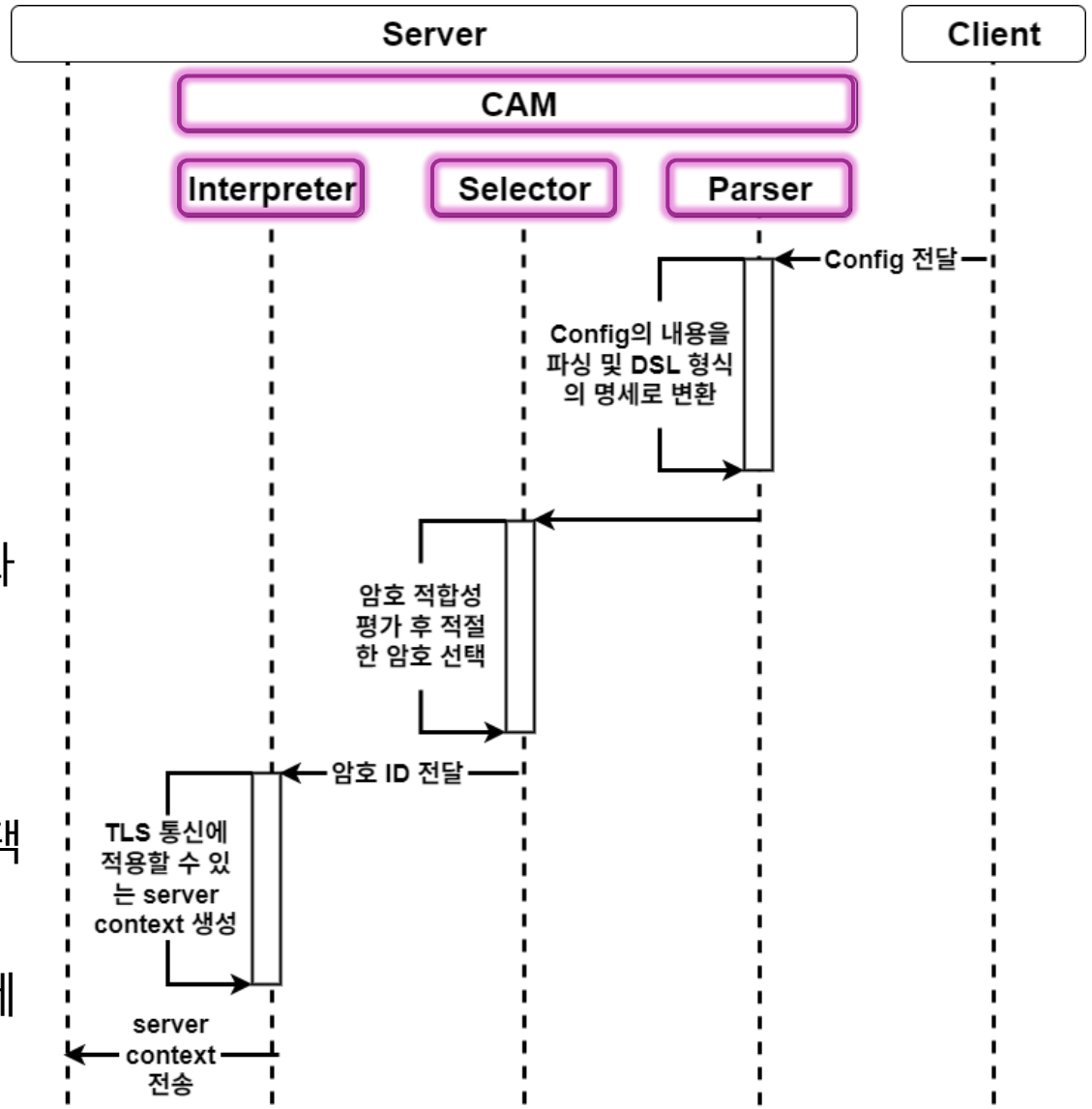
- **NIST 관점에서의 Crypto Agility 시스템 구현을 위한 추가 요구사항 도출**
  - Centralized policy control을 위한 descriptive context 및 인터페이스 의 정의가 필요
- **사용자 정책 기반 미들웨어 프레임워크(CAMF) 제안**
  - Domain-specific Language (DSL) 기반의 시멘틱을 정의 후, 이를 기반으로 하여 자동으로 적용하거나 교체할 수 있게 하기 위한 암호 정책 descriptor(PolicySemantics)를 정의
- **PQ-TLS 확장 구조 설계 및 사례 연구**
  - CAMF를 기반으로 NIST PQC와 KpqC 선정 알고리즘, 그리고 기존 공개키 암호를 동시에 지원하는 PQ-TLS 확장 구조를 설계 및 사례 연구

미들웨어 프레임워크 설계

[Crypto Agility Middleware Framework, CAMF]

- Client
  애플리케이션 특성에 따라 암호 모듈 사용 정책 작성·수정
- Server
  서비스 제공자, 제어 정책과 암호문 송수신 분리
  Crypto Agility Manager(CAM)를 구동하는 호스트

1) Client → Server: 요구사항 명세(Config)를 ClientHello와 함께 전송
2) CAM 처리
   2-1) Parser: config → DSL 변환
   2-2) Selector: 적합한 backend 암호 모듈/알고리즘 선택
   2-3) Interpreter: TLS server context로 변환
3) Server → Client: 선택한 암호를 기반으로 context와 함께 TLS handshake 진행



미들웨어 프레임워크 설계

[Domain-Specific Language (DSL) 설계]

- CAM이 정책·환경 변화에 따라 적절한 알고리즘을 자동 선택하려면 처리 가능한 정책 시멘틱의 정의가 필요

> *A computer programming language of limited expressiveness focused on a particular domain.*
>
> Fowler, Martin (2010). *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

- **Solution: CAMF 전용 DSL 설계**
  - Requirement 1: 보안 이벤트, 파라미터, 보안 수준 명확히 기술
  - Requirement 2: 이질성을 지원하기 위해 다양한 암호 primitive · 알고리즘 명시가 가능해야 함
  - Requirement 3: 상호운용·호환성 확보를 위해 자원 상태 · 플랫폼 · OS · HW 정보 등 운영 환경에 대한 정보가 필요
  - 일반적으로, XML 또는 YAML과 같은 textual data structure representation으로 표현

미들웨어 프레임워크 설계

[Domain-Specific Language (DSL) 설계]

- **PolicySemantics**
  - 실행 환경 조건(메모리, 네트워크 단편화, 지연 시간 등)을 기반으로 자동 알고리즘 선택/교체 로직에서 활용

  - enabled_filters: 보안·지연·단편화 조건 설정
  - runtime_selection: 상황별 우선순위 적용
    - fragmentation_sensitive – 패킷 수
    - prefer_low_latency – 지연 시간

  - TLS 1.3 예시: ClientHello에 자원 정보·보안수준·가정 전달 (e.g., YAML/XML) → 서버에서 PolicySemantics 생성

※CPU 사양 등 환경 요소를 추가해 정밀 최적화 가능

```yaml
# PolicySemantics
policy:
  id: "policy_pq_runtime_01"
  description: >
    Runtime PQ scheme selection based on user-enabled filters
    (security level, latency) and default resource constraints.
  conditions:
    enforce_strict_nist_level: 3
    allowed_assumptions: ["MLWE", "RLWE"]
    require_hybrid:
      kem: true
      signature: true

  resource_constraints:
    memory_limit_mb: 512           # MB

    latency_budget_ms:
      server: 10
      client: 10

    runtime_selection:
      fragmentation_sensitive: true
      prefer_low_latency: true

  enabled_filters:
    - security
    - latency
```

미들웨어 프레임워크 설계

[Domain-Specific Language (DSL) 설계]

- **KeyContext**
  - PolicySemantics 조건을 만족하는지 판단하기 위한 메타데이터 집합
  - 정책 조건과 성능 지표를 연동해 실현 가능한 스킴만 선택
  - nist-level, assumption(안전성 가정)
  - keygen/encapsulation/decapsulation_latency
  - 활용 예시: 클라이언트 측 지연 허용 시간(latency_budget_ms)과 KeyContext 값 비교 제한 범위 내 후보만 선별

```yaml
# KeyContext
key_context:
  kem:
    name: MLKEM768
    nist_level: 3
    assumption: MLWE
    pk_size: 1184
    ct_size: 1088
    keygen_latency: 0.0151          # ms
    encapsulation_latency: 0.0194   # ms
    decapsulation_latency: 0.0152   # ms
  signature:
    name: AIMer192f
    nist_level: 3
    assumption: AIM
    sig_size: 13056
    sign_latency: 17.8454          # ms
    verify_latency: 1.126          # ms
```

미들웨어 프레임워크 설계

[Domain-Specific Language (DSL) 설계]

- **CryptoModuleContext**
  - 시스템에서 사용 가능한 암호 라이브러리/모듈의 지원 목록 관리 (e.g., 각 라이브러리(liboqs, openssl)가 제공하는 KEM/DSA 알고리즘 목록)

  - CAM이 실제 환경에서 활용 가능한 모듈 선택·교체하는 데 기준 데이터베이스 역할

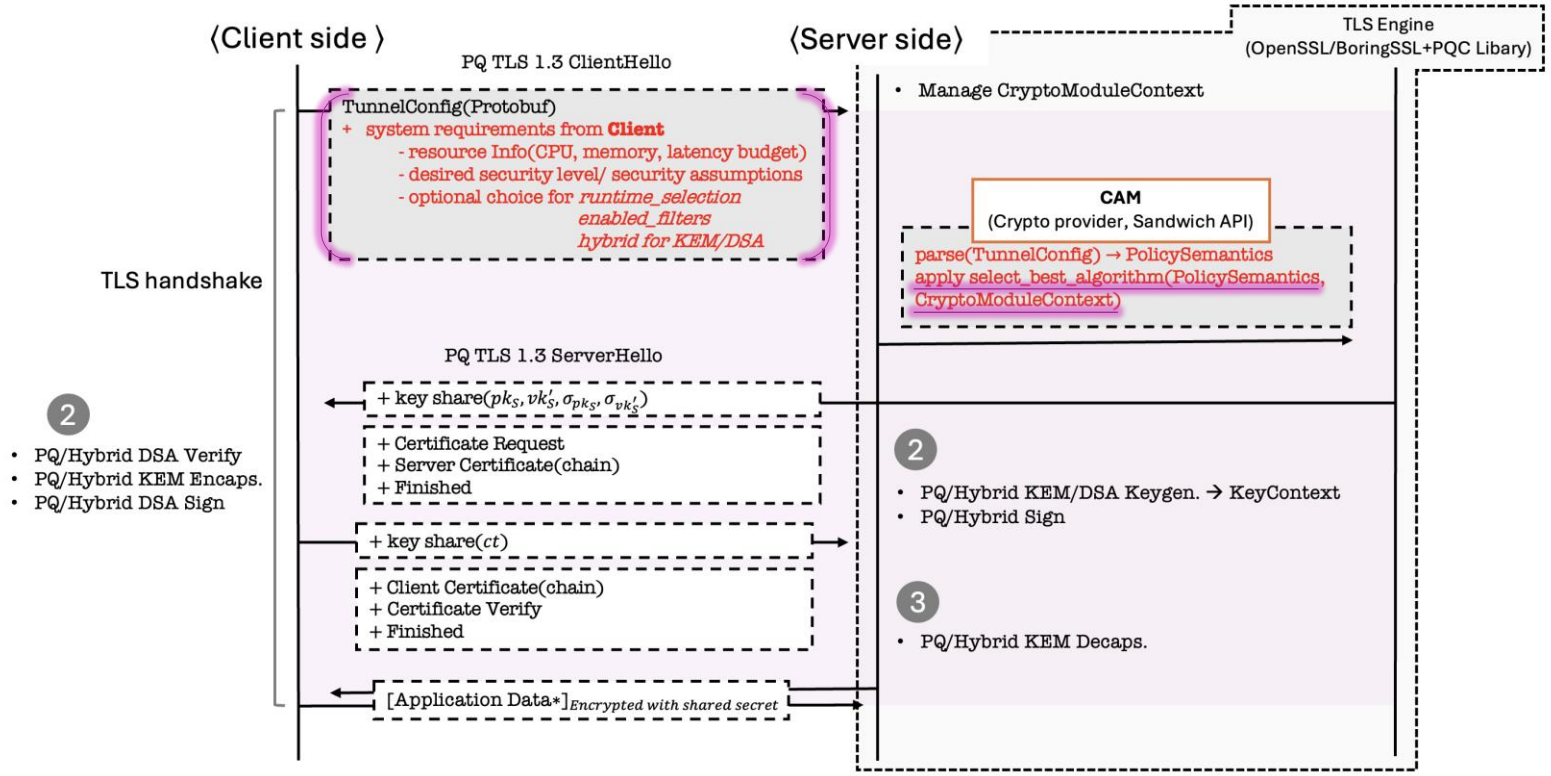
"Interoperability가 목적"
- 정책 기반으로 모듈 전환 가능
- 상황 변화 시 알고리즘·라이브러리 동적 교체 지원

```yaml
crypto_module_context:
  backend:
    - name: liboqs
      supported_kems: [ML_KEM-512, HQC-128, ...]
      supported_signatures: [ML_DSA-65, ...]
    - name: openssl
      supported_kems: [ecdhe_secp256r1, ...]
      supported_signatures: [ecdsa_p256, ...]
```

활용 시나리오 사례 연구

[CAMF 기반 TLS handshake workflow]

- 기존 TLS handshake과의 차이점
  - ClientHello시 정책 기반 요구사항(config) 포함
  - 서버 측 CAM이 파싱 후 PolicySemantics 생성 → 알고리즘 선택 함수 호출
  - TLS Server Context 생성 전 최적의 PQ 또는 hybrid 알고리즘 결정 후 이를 handshake에 반영



활용 시나리오 사례 연구

[클라이언트 정책 명세 기준 예시: 안전성 가정]

Schemes	Security assumption
NTRU+	NTRU, RLWE
SMAUG-T	MLWE, MLWR
MLKEM	MLWE
HQC	SDP

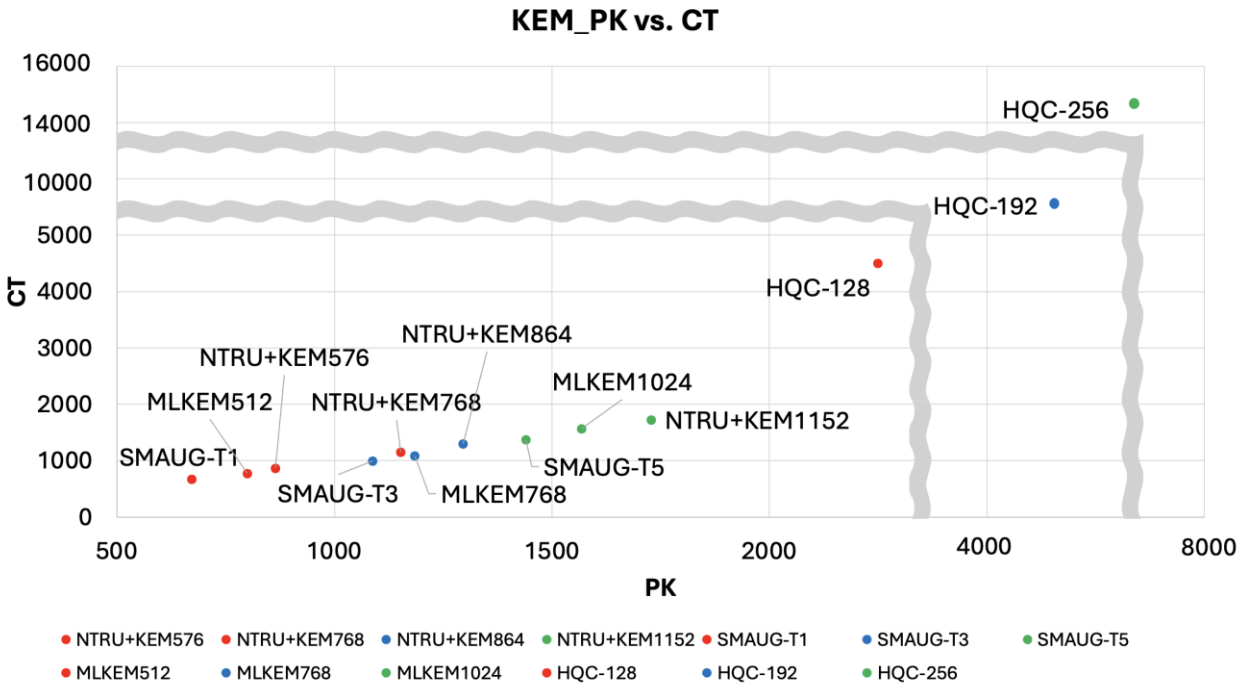
<KEM 안전성 가정>

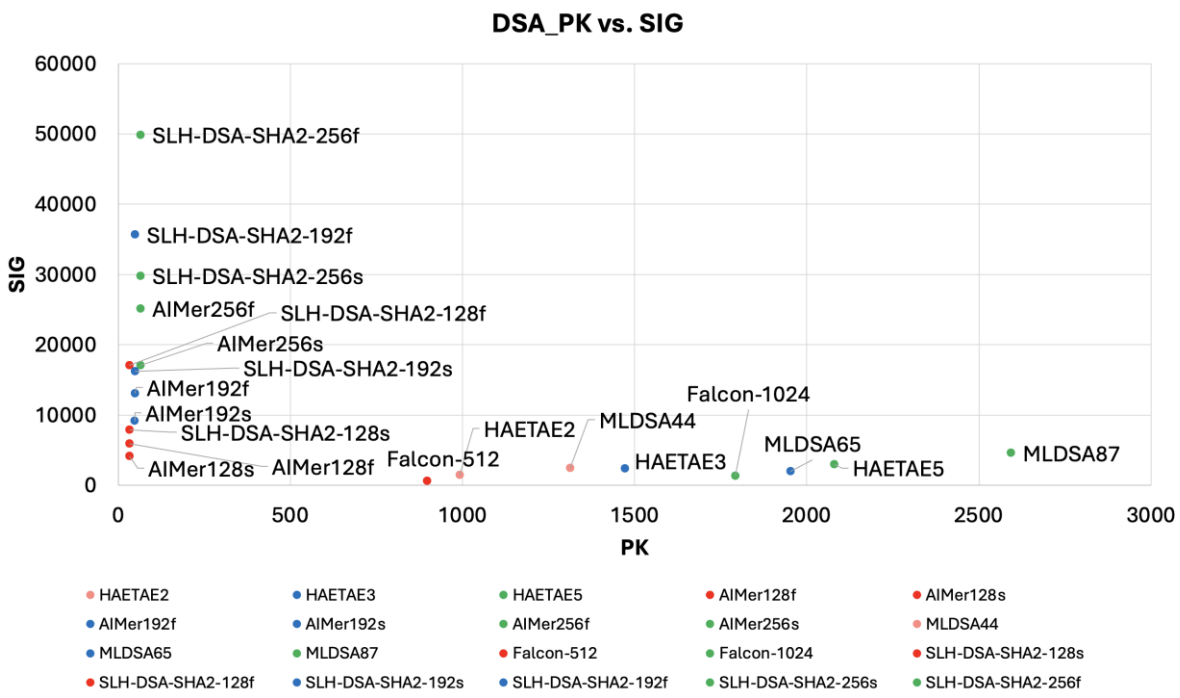
Schemes	Security assumption
HAETAE	MLWE, MSIS
AIMer	AIM
MLDSA	MLWE, MSIS
Falcon	NTRU
SLH-DSA	SHAKE or SHA2

<DSA 안전성 가정>

활용 시나리오 사례 연구

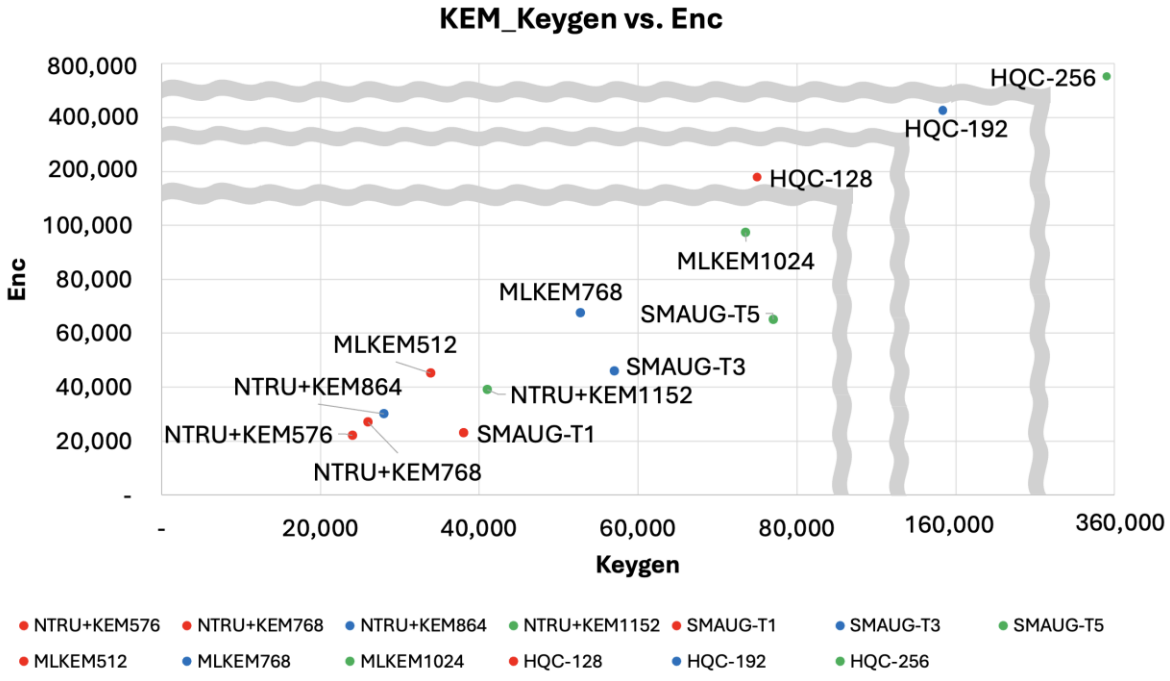
[클라이언트 정책 명세 기준 예시: Size]

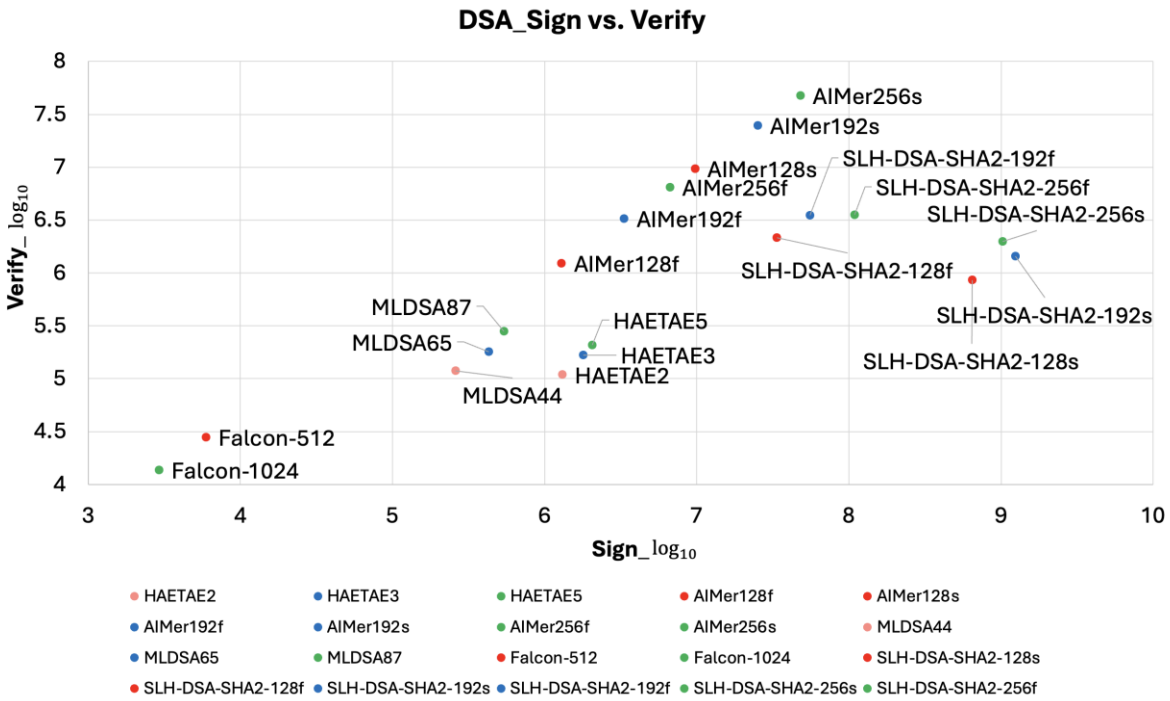




활용 시나리오 사례 연구

[클라이언트 정책 명세 기준 예시: Latency]





활용 시나리오 사례 연구

[PolicySemantics 명세 예시: 안전성 우선 시나리오]

- 성능 < 안전성
- 클라이언트는 NIST가 정의한 보안 등급 레벨 중 **5** 이상을 만족하는 알고리즘만을 허용 **(enforce_strict_nist_level)**
- 안전성 가정: MLWE, MSIS, AIM, SDP 기반의 스킴만을 신뢰 가능한 후보로 간주 **(allowed_assumptions)**
- 정책 수준에서 보안 필터만을 활성화 **(enabled_filters)**
- 지연시간에 대한 우선 순위 적용 **(prefer_low_latency)**

```yaml
# Scenario 1
policy_1:
  id: "policy_pq_runtime_01"
  description: >
    Runtime PQ scheme selection based on user-enabled filters
    (security level) and default resource constraints.
  conditions:
    enforce_strict_nist_level: 5
    allowed_assumptions: ["MLWE", "MSIS", "AIM", "SDP"]
    require_hybrid:
      kem: false
      signature: false

  resource_constraints:
    memory_limit_mb: 512           # MB

    latency_budget_ms:
      server: 0.1
      client: 0.1

    runtime_selection:
      fragmentation_sensitive: false
      prefer_low_latency: true

  enabled_filters:
    - security
```

활용 시나리오 사례 연구

[PolicySemantics 명세 예시: 성능 우선 시나리오]

- 성능 > 안전성
- 클라이언트는 NIST가 정의한 보안 등급 레벨 중 1 이상을 만족하는 알고리즘이면 허용 **(enforce_strict_nist_level)**
- 실시간성 보장을 위해 서버/클라이언트 latency budget이 constraint으로 작용 **(latency_budget_ms)**
- 안전성 가정: MLWE, MSIS, AIM, SDP 기반의 스킴만을 신뢰 가능한 후보로 간주 **(allowed_assumptions)**
- 정책 수준에서 latency 필터만을 활성화 **(enabled_filters)**
- 지연시간(prefer_low_latency) 값이 런타임 암호 선택에 관여

```yaml
# Scenario 2
policy_2:
  id: "policy_pq_runtime_02"
  description: >
    Runtime PQ scheme selection based on user-enabled filters
    (latency) and default resource constraints.
  conditions:
    enforce_strict_nist_level: 1
    allowed_assumptions: all
    require_hybrid:
      kem: false
      signature: false

  resource_constraints:
    memory_limit_mb: 256            # MB

    latency_budget_ms:
      server: 0.05
      client: 0.01

    runtime_selection:
      fragmentation_sensitive: false
      prefer_low_latency: true

  enabled_filters:
    - latency
```

Conclusion

- **Summary**
  - 사용자 정책 기반의 미들웨어 프레임워크를 NIST 관점의 암호 민첩성 실현할 수 있는 하나의 방법론으로서 제안

- **Further Work**
  - Sandwich API 코드 분석 및 TLS Server/Client handshake 동작 확인(Cont.)
  - 자동으로 crypto module을 교체가 이루어지기 위한 미들웨어 프레임워크 PoC 구현