POLITECNICO DI TORINO

Master's degree course in Cybersecurity

Master's Degree Thesis

Post-Quantum IPsec Gateway: Policy Enforcement Point

Supervisors
Prof. Antonio Lioy
Dott. Flavio Ciravegna

Candidate
Leonardo RIZZO

Academic Year 2024-2025

Summary

The emergence of large-scale quantum computing poses a significant threat to the Public Key Infrastructure (PKI) that secures modern communications. Protocols such as Internet Protocol Security (IPsec), which rely on Internet Key Exchange version 2 (IKEv2) for key establishment, are fundamentally vulnerable to Shor's algorithm. This vulnerability creates an immediate Harvest Now, Decrypt Later (HNDL) attack vector, where encrypted data harvested today can be retrospectively decrypted once a sufficiently powerful quantum computer is available.

While the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standardisation process has produced new quantum-resistant algorithms, a simple "rip and replace" migration strategy is untenable. The volatility of new cryptographic assumptions, exemplified by the catastrophic failure of SIKE and the practical threat of implementation-specific Side-Channel Attacks (SCAs), demands a new architectural paradigm: cryptographic agility.

This thesis presents the design, implementation, and evaluation of a PQC-Agile IPsec Gateway. The core contribution is a novel architecture that decouples cryptographic policy enforcement from the protocol's core logic. The solution leverages strongSwan as a high-performance Policy Enforcement Point (PEP) and Open Policy Agent (OPA) as a centralised, declarative Policy Decision Point (PDP).

The strongSwan ext-auth plugin is modified and patched to intercept IKEv2 negotiations, gathering the peer's cryptographic proposal and certificate metadata. This context is sent as a structured JSON query to the OPA engine, which evaluates it against fine-grained Rego policies. These policies enforce minimum security levels (KE and SIG) and validate a diverse cryptographic portfolio-including Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM), Module-Lattice-Based Digital Signature Algorithm (ML-DSA), Bit-Flipping Key Encapsulation (BIKE), and Hamming Quasi-Cyclic (HQC), to provide resilience against the failure of a single algorithm family.

Policy decisions are applied through asynchronous helper services, using strongSwan's VICI interface to manage the Child Security Association (CHILDSA) lifecycle.

The result is a resilient and operationally flexible gateway that supports auditable and incremental migration to PQC. Administrators can dynamically update cryptographic policy, disable compromised algorithms, or enforce hybrid deployments in real time, without service interruption or gateway redeployment.

Acknowledgements

This thesis project was conducted in collaboration with my colleague, Simone Sambataro.

Contents

Chapter 1

Introduction

1.1 The Dawn of the Quantum Threat

The fabric of modern digital security is woven from complex mathematical problems, long believed to be intractable for classical computers. Our entire ecosystem of secure communication, ranging from the protocols that protect web traffic, such as Transport Layer Security (TLS), to the digital signatures that verify software and the Virtual Private Networks (VPNs) that secures corporate access, is built upon this foundation [1]. Specifically, the security guarantees of most contemporary Public-Key Cryptography (PKC) systems, including industry cornerstones like RSA and Elliptic-Curve Cryptography (ECC), depend directly on the computational difficulty of two core problems: integer factorisation and the discrete logarithm problem, respectively. The asymmetric nature of these problems, where they are easy to compute in one direction but extremely difficult to reverse, has provided robust assurance of data confidentiality, integrity, and authenticity for decades.

This long-standing security paradigm, however, is facing an existential challenge. The emergence of quantum computing marks a fundamental shift in computational power. Unlike classical bits, which exist as either a 0 or a 1, a quantum bit (or 'qubit') can leverage the quantum-mechanical principles of superposition and entanglement to represent multiple states simultaneously. This capability allows a quantum computer to perform parallel computations on a massive scale, moving certain types of problems from the realm of the computationally intractable to the feasible [2].

This is not merely a theoretical exercise. In 1994, a seminal paper by Peter Shor demonstrated an algorithm that, if run on a sufficiently large-scale and stable quantum computer, could solve both integer factorisation and the discrete logarithm problem in polynomial time [3]. The implications of this discovery are profound: a machine capable of executing Shor's algorithm would render the vast majority of our current public-key infrastructure obsolete. Security guarantees that would take a classical computer millennia to break could be compromised in a matter of hours or days.

1.2 The "Harvest Now, Decrypt Later" Urgency

While the precise timeline for the construction of such a 'cryptographically relevant' quantum computer remains a subject of intense debate, the threat possesses an urgency that is independent of that timeline. This urgency is defined by the HNDL attack vector [4]. In this scenario, an adversary can intercept and store large volumes of currently encrypted data today. Even if this data is protected by strong, classical cryptography, the adversary can simply hold it, waiting for the day a viable quantum computer becomes available. At that point, they can retrospectively decrypt years' worth of sensitive, captured information [1].

This HNDL threat model creates an immediate and critical problem for any infrastructure that protects long-lived, sensitive information. Government communications, corporate intellectual property, medical records, and critical infrastructure commands, all of which are frequently

secured by VPN tunnels, are prime targets. Data that must remain confidential for 10, 20, or 50 years is already at risk of being harvested.

1.3 Problem Statement and Thesis Goal

This specific vulnerability directly impacts the core protocols of modern secure networking. The IPsec protocol suite, which forms the backbone of most VPNs, relies on the IKEv2 protocol for key exchange and mutual authentication. These IKEv2 negotiations were designed around ECC and RSA-based mechanisms and are therefore fundamentally vulnerable to both Shor's algorithm and the HNDL attack scenario. It is consequently imperative to begin migrating these critical network gateways to quantum-resistant standards.

However, a simple "rip and replace" migration is not a viable strategy. The PQC landscape is still in flux; the NIST standardisation process is ongoing, new algorithms are being finalised, but existing ones may yet be found to have weaknesses or performance issues. This uncertainty demands a more sophisticated, flexible, and forward-looking approach.

The goal of this thesis is therefore to design, implement, and evaluate a **Post-Quantum Crypto-Agile Gateway**. This solution aims to provide a robust IPsec endpoint, based on the well-regarded strongSwan platform, that not only implements emerging PQC algorithms but also possesses the **agility** to adapt to changing cryptographic standards dynamically. This agility will be achieved by externalising cryptographic policy decisions from the gateway's core logic to a dedicated, centralised policy engine. This architecture allows administrators to enforce granular rules (e.g., mandating PQC for high-security users, permitting classical algorithms for legacy clients, or immediately disabling a newly compromised algorithm) without service disruption or complex reconfiguration of the gateway itself.

1.4 Thesis Structure

To achieve this goal, this thesis is structured as follows. Chapter 2 provides a comprehensive review of the background and state of the art. It covers the PQC landscape and the NIST standardisation process, delves into the IPsec and IKEv2 protocols and their new PQC-related extensions, and introduces the key technologies used in our solution, such as the strongSwan platform and the OPA. Chapter 3 (insert ref here) details the architectural design of our proposed crypto-agile gateway, focusing on the separation of concerns and the interaction between the PEP and PDP components. Chapter 4 (insert ref here) presents the practical implementation of the system, detailing configuration, policy-writing, and the validation of key use cases. This chapter also includes a quantitative analysis of the system's performance, specifically focusing on connection latency and policy-decision overhead. Finally, Chapter 5 (insert ref here) concludes the thesis by summarising the results achieved, discussing the limitations of the current work, and proposing potential avenues for future research and development.

Chapter 2

State of the Art

As introduced in the previous chapter, the emergence of quantum computing threatens to render the entire infrastructure of modern cryptographic security obsolete. To fully comprehend the nature of this threat and, consequently, the foundations of PQC, it is essential to analyse not only the cryptographic protocols themselves but also the quantum algorithms that undermine their security. This chapter analyses the state of the art of the technologies fundamental to this thesis.

We begin by establishing the quantum threat landscape, before moving to the networking technologies (IPsec) and software platforms (strongSwan, OPA) that form the basis of our solution.

2.1 The Post-Quantum Cryptography Landscape

The field of PQC was not born in a vacuum; it is a direct and necessary response to the capabilities demonstrated, at least theoretically, by quantum algorithms. The quantum threat is not monolithic: it impacts different types of cryptography (symmetric and asymmetric) in different ways and with varying degrees of severity.

2.1.1 The Quantum Threat and Foundational Algorithms

For decades, public-key cryptography has relied upon the computational difficulty of problems such as the factorisation of large integers or the calculation of the discrete logarithm [5]. These problems are considered intractable for classical computers, meaning the time required to solve them grows exponentially with the size of the input (the key length). Quantum algorithms, however, operate under a different computational model, leveraging the principles of quantum mechanics to achieve computational advantages that, in some cases, prove to be exponential.

Shor's Algorithm: The Asymmetric Cryptography Breaker

The algorithm that initiated the race toward PQC was published by Peter Shor in 1994 [3]. The impact of this work cannot be overstated, as it demonstrated that a sufficiently powerful quantum computer would be capable of solving the two mathematical problems that underpin all modern public-key cryptography in polynomial time:

1. **Integer Factorisation Problem (IFP):** This is the problem upon which the security of the RSA algorithm is based. Given an integer N, find its prime factors p and q.
2. **Discrete Logarithm Problem (DLP):** This is the problem that underpins the Diffie-Hellman (DH) key exchange and the ECC algorithm.

Whilst the best-known classical algorithms, such as the General Number Field Sieve (GNFS), require super-polynomial (almost exponential) time to solve these problems, Shor's algorithm provides a solution in polynomial time, $O((\log N)^3)$. It achieves this by leveraging the power of the *Quantum Fourier Transform* (QFT) to find the period of a function, which can then be used to determine the factors of N [5].

The impact is catastrophic: the entire ecosystem of TLS, digital signatures, PKIs, and, most critically for the context of this thesis, IKEv2 key exchanges based on DH and ECC, would be completely compromised. It is this exponential speed-up that makes Shor's algorithm the primary threat and the main driver of PQC standardisation.

Grover's Algorithm: The Symmetric Cryptography Accelerator

The second major algorithmic threat, albeit of a different nature, is Lov Grover's algorithm from 1996 [6]. Unlike Shor's, Grover's algorithm does not attack the underlying mathematical structure of a problem but instead focuses on a much more generic task: searching an unstructured database.

A classical brute-force attack against, for example, a 128-bit Advanced Encryption Standard (AES) key requires, on average, 2^{127} attempts to find the correct key within a search space of $N = 2^{128}$ elements. Grover's algorithm dramatically reduces this time.

It operates as a process of *amplitude amplification*. Starting from a uniform superposition of all possible states (all possible keys), the algorithm iteratively applies an "oracle" operation (which "marks" the correct solution) and a "diffusion" operation (which amplifies the probability amplitude of the marked state). The result is that Grover's algorithm can find the target item in only $O(\sqrt{N})$ steps [6, 5].

The impact, in this case, is not an exponential speed-up, but "only" a quadratic one. Nevertheless, the consequences are significant:

- An AES-128 key, which offers 128 bits of security against a classical computer, provides only 64 bits of security against a quantum computer running Grover's algorithm (since $2^{64} \approx \sqrt{2^{128}}$).
- An AES-256 key, on the other hand, provides $256/2 = 128$ bits of quantum security.

This means that whilst Grover's algorithm weakens symmetric cryptography, it does not "break" it in the way Shor's breaks PKC. The countermeasure is straightforward and already available: double the symmetric key length. For this reason, PQC standards (including NIST) recommend the use of AES-256 to guarantee long-term security (at least 128-bit) in the post-quantum era.

Brassard et al.: Amplitude Amplification and Counting

Grover's algorithm is not an isolated case, but rather the most famous example of a broader class of algorithms based on *Amplitude Amplification*. Gilles Brassard and his colleagues generalised Grover's technique in a foundational work [7].

Whereas Grover's algorithm applies to a search where a single (or known number of) solution exists, Quantum Amplitude Amplification (QAA) is a more general framework. It can be applied to any heuristic quantum algorithm that has some probability p of success. QAA can "boost" this algorithm to find the solution in a time proportional to $1/\sqrt{p}$, greatly enhancing its efficiency [7].

A direct application of this technique is *Quantum Counting*, which allows one to not only find an item, but to estimate **how many** marked items exist in the database. This has implications for more sophisticated cryptanalytic attacks, such as collision attacks, which are the basis for attacks on hash functions (like SHA-256). Here too, the impact is quadratic, meaning that SHA-256 (offering 128-bit collision security) is still considered sufficiently robust for post-quantum use.

In summary, whilst Shor's algorithm devastates asymmetric cryptography, the impact of Grover and Brassard on symmetric cryptography and hash functions is manageable and requires a simple (but mandatory) increase in security parameters. Our gateway architecture will therefore need to account for both of these threats.

2.1.2 The NIST PQC Standardization Process

Recognising the profound and existential threat posed by Shor's algorithm, the U.S. National Institute of Standards and Technology (NIST) took a proactive and globally-leading role. Rather than waiting for the threat to materialise, NIST initiated an open, collaborative, and transparent process to solicit, evaluate, and standardise a new suite of quantum-resistant public-key cryptographic algorithms. This process formally began in 2016 with a public call for proposals [8].

This was not a typical standardisation effort; it was effectively a global competition, drawing submissions from leading cryptographers, academic institutions, and private sector researchers worldwide. The goal was to identify and standardise algorithms for two primary functions:

- **Public-Key Encryption (PKE) and Key Encapsulation Mechanisms (KEMs):** Algorithms to replace Diffie-Hellman and ECC for establishing shared secrets over insecure channels.
- **Digital Signatures:** Algorithms to replace RSA and ECDSA for verifying authenticity and integrity.

Evaluation Criteria and Security Levels

NIST did not simply ask for 'secure' algorithms; it defined a rigorous set of evaluation criteria, prioritising security above all else, followed by performance and implementation characteristics [8].

The most critical part of this framework was the definition of **security categories**. Instead of relying on abstract complexity, NIST benchmarked the required quantum-resistance against the known costs of breaking established symmetric algorithms using Grover's algorithm (as discussed in Section 2.1.1). This created five distinct security levels:

- **Level 1:** At least as hard to break as finding an AES-128 key using Grover's algorithm (approx. 2^{64} quantum operations).
- **Level 2:** At least as hard to break as finding a SHA-256 collision using Grover's (approx. 2^{128} quantum operations).
- **Level 3:** At least as hard to break as finding an AES-192 key using Grover's (approx. 2^{96} quantum operations).
- **Level 4:** At least as hard to break as finding a SHA-384 collision using Grover's (approx. 2^{192} quantum operations).
- **Level 5:** At least as hard to break as finding an AES-256 key using Grover's (approx. 2^{128} quantum operations).

It is important to note the apparent non-linearity: Level 2 and Level 5 both correspond to a 2^{128} computational cost. However, Level 2 refers to the cost of a pre-quantum attack on a (hypothetical) algorithm, whilst Level 5 refers to the cost of a quantum attack (Grover's) on AES-256. In practice, the process focused primarily on achieving security at **Levels 1, 3, and 5**, which correspond to the quantum security of AES-128, 192, and 256, respectively.

The Competing Cryptographic Families

The 69 submissions accepted into Round 1 of the NIST process were not built on arbitrary principles. They were derived from a handful of mathematical families whose underlying problems are believed to be resistant to quantum attacks [9]. The entire standardisation process was, in effect, a "battle" between these families, each with distinct performance trade-offs.

Lattice-based Cryptography: This family is based on the difficulty of problems on "lattices", such as the Learning With Errors (Learning With Errors (LWE)) problem. These algorithms emerged as the clear front-runners, offering a balanced "all-rounder" profile: strong security proofs, relatively small key/signature sizes, and very fast computation. **Kyber** and **Dilithium** are the most prominent examples.

Code-based Cryptography: This is one of the oldest PQC families, based on the McEliece cryptosystem (1978). Its security relies on the difficulty of decoding a general linear error-correcting code. Its primary advantage is its long history and confidence in its security. Its major disadvantage, however, is the very large size of the public keys, often measured in hundreds of kilobytes or even megabytes.

Hash-based Cryptography: This family builds security relying **only** on the properties of a secure cryptographic hash function (like SHA-256). This is a significant advantage, as its security is well-understood and not dependent on novel mathematics. Its main drawback is that signatures are either "stateful" (requiring the signer to remember which state was used, which is dangerous if not handled perfectly) or "stateless" (like **SPHINCS+**), which results in very large signatures and slow signing operations.

Multivariate Cryptography: This family bases its security on the difficulty of solving systems of non-linear (multivariate) polynomial equations over a finite field. These schemes often feature very small signatures, but many candidates proved difficult to design securely, with several prominent submissions being broken during the NIST process.

Isogeny-based Cryptography: This was once a promising family that uses the properties of "isogenies" (maps between elliptic curves). Its main advantage was the potential for very small key sizes, reminiscent of classical ECC. However, the leading Key Encapsulation Mechanism (KEM) candidate in this family (SIKE) was dramatically and practically broken in 2022, effectively removing this family from the main competition and highlighting the volatility of the PQC landscape.

This competition between families explains NIST's final selection. They chose the high-performance "all-rounders" from the lattice family as the primary standards, while also selecting a hash-based scheme as a more conservative, well-understood backup, should any systemic flaw in lattices be discovered.

The Selection Rounds and Finalised Standards

The standardisation process was structured as a multi-round "tournament" designed to progressively filter the submissions through intense public cryptanalysis, informed by the trade-offs of the families discussed above.

- **Round 1 (2017-2018):** Began with 69 initial valid submissions. This phase focused on initial security analysis, and many submissions were broken or found to be insecure.
- **Round 2 (2019-2020):** Advanced with 26 candidates, focusing on deeper analysis and initial performance benchmarking.
- **Round 3 (2020-2022):** Focused on 7 "finalists" and 8 "alternates". This phase involved intensive implementation, performance testing on various platforms (from servers to microcontrollers), and deep cryptanalysis.

In July 2022, NIST announced its landmark decision, selecting the first four algorithms for standardisation [10]. This selection was finalised in August 2024 with the publication of the official Federal Information Processing Standards (FIPS) drafts. The selected algorithms are summarised in Table 2.1.

Table 2.1. Finalised NIST PQC Standards (as of August 2024)

Standard	Algorithm Name	Function	Cryptographic Family
FIPS 203	**ML-KEM** (Kyber)	KEM / PKE	Lattices (LWE)
FIPS 204	**ML-DSA** (Dilithium)	Signature	Lattices (MLWE)
FIPS 205	**SLH-DSA** (SPHINCS+)	Signature	Hash-Based

This selection highlights the dominance of lattice-based cryptography, which forms the basis for the primary KEM and signature standards.

Primary Standards (FIPS 203 & 204): For general-purpose use, NIST selected two primary algorithms: **CRYSTALS-Kyber** (standardised as ML-KEM) and **CRYSTALS-Dilithium** (standardised as ML-DSA) [11, 12]. Both are based on the difficulty of problems over module lattices. They were chosen for their excellent all-around performance, offering small key and signature sizes combined with fast operation, making them suitable for a wide range of applications, including network protocols like IKEv2.

Additional Standard (FIPS 205): Recognising the need for diversity, NIST also standardised **SPHINCS+** (as SLH-DSA) [13]. SPHINCS+ is a stateless hash-based signature scheme. Its primary advantage is that its security is not based on novel mathematical problems (like lattices) but relies only on the security of the underlying hash function. This provides a valuable, albeit slower, alternative in the event that unforeseen weaknesses are discovered in lattice-based cryptography.

The Fourth Round (Ongoing KEMs): NIST also initiated a Fourth Round to further evaluate additional KEM candidates, primarily to find algorithms based on different mathematical principles (code-based, isogeny-based) for greater cryptographic diversity. This indicates that whilst the first standards are finalised, the PQC landscape continues to evolve, a fact that directly motivates the need for the crypto-agile gateway proposed in this thesis.

2.1.3 Foundations of Key Cryptographic Families

The standards selected by NIST (Table 2.1) provide the primary approved algorithms, but they do not represent the entire state of the art. A critical aspect of the PQC transition is "cryptographic diversity", the strategy of relying on multiple, mathematically distinct hard problems to hedge against the risk that an unforeseen breakthrough could weaken an entire family (such as lattices).

The NIST process itself championed this diversity by advancing algorithms from different families to the final round. By understanding the foundations of the lattice-based winners, alongside the most prominent code-based finalists, is essential to grasp the trade-offs available for building a truly resilient crypto-agile system.

Lattice-Based Standards (ML-KEM and ML-DSA) The core of the modern PQC landscape is built upon the two primary NIST standards, ML-KEM (Kyber) [11] and ML-DSA (Dilithium) [12]. These are "sister" algorithms founded on the difficulty of the **Module Learning With Errors (MLWE)** problem, an efficient variant of the foundational **LWE** problem [14].

- **ML-KEM (Kyber):** This IND-CCA2-secure KEM is designed to directly replace the Diffie-Hellman key exchange. The receiver's public key defines a system of "noisy" equations over a module lattice. The initiator encapsulates a shared secret using this public key, creating a ciphertext. Due to the injected noise, only the receiver (holding the private key, or the "solution" to the equations) can remove the noise and decapsulate the identical shared secret.
- **ML-DSA (Dilithium):** This signature scheme uses the Fiat-Shamir transform with "rejection sampling." The private key consists of statistically "short" vectors (polynomials). To sign a message, the signer must produce a signature that is also verifiably "short" and solves a public equation. This "proof of shortness" is computationally infeasible without the private key.

Code-Based Diversity (HQC and BIKE) To provide a robust alternative to the lattice-based family, the NIST process also identified code-based KEMs as strong candidates. These algorithms derive their security from the difficulty of decoding error-correcting codes, a problem that is mathematically unrelated to lattices. Two of the most prominent finalists in this category were HQC and BIKE.

- **HQC (Hamming Quasi-Cyclic):** HQC is a code-based KEM that builds on the principles of the McEliece cryptosystem [15]. Its security is based on the difficulty of decoding a random quasi-cyclic code, a well-understood problem in cryptography.
- **BIKE (Bit-Flipping Key Encapsulation):** BIKE is also a code-based KEM, but it relies on a different class of codes: **Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC)** codes [16]. Its name derives from its efficient decapsulation process, which uses an iterative "bit-flipping" decoder.

The existence of these distinct and viable cryptographic families (lattices, codes) provides the building blocks for a truly crypto-agile architecture. The design choices are no longer limited to a single algorithm, but can instead involve creating hybrid or composite schemes that leverage the strengths of each, a core concept that will be explored in the design of our gateway.

2.1.4 The Challenge of a Post-Quantum PKI

While the finalisation of these standards [11, 12, 13] provides the necessary cryptographic primitives, it simultaneously introduces significant engineering challenges for integration. The most acute of these, directly impacting the `IKE_AUTH` exchange [17], is the incompatibility of the new signature schemes with the existing X.509 PKI.

The core of this challenge lies in the disparity in size. Classical public keys (like ECDSA P-256) and their signatures are measured in tens of bytes. In contrast, the new PQC public keys and signatures are orders of magnitude larger. For example, an "ML-DSA-44" public key is over 1.3 KB, while an "SLH-DSA" (SPHINCS+) signature can be 8 KB or larger, depending on the security level [12, 13].

This size increase directly conflicts with the assumptions baked into the X.509 PKI standard [18], which was not designed to embed multi-kilobyte public keys. This, in turn, creates a significant challenge for network protocols like IKEv2, which were not designed to transport multi-kilobyte `CERT` and `AUTH` payloads [17]. As will be discussed in detail in Section 2.4, this size problem leads to critical, practical networking issues such as IP fragmentation, which can cause interoperability failures.

To address this challenge and to hedge against future breaks, the IETF is standardising mechanisms for **composite** or **hybrid** authentication [19]. In this model, a single certificate or signature payload contains *both* a classical primitive (e.g., "Ed448") and a PQC primitive (e.g., 'ML-DSA-87'). A verifier must validate *both* signatures to accept the authentication.

This composite approach is the precise mechanism that underpins the signature-level policies (such as "SIG-L3-SUF") discussed in this thesis's design. It illustrates that migration is not a simple algorithm swap. This complex, multi-layered authentication landscape creates another powerful driver for crypto-agility, as gateways must be able to parse, validate, and enforce policies on these new hybrid structures.

2.1.5 The Need for Crypto-Agility

The finalisation of the first FIPS standards by NIST does not represent an end-point, but rather the beginning of a long and complex migration. The history of cryptography has taught us that statically embedding algorithms into protocols and infrastructure, a practice known as "hard-coding", is a critical vulnerability. The transition to PQC is not a simple "rip and replace" operation; it is a dynamic and uncertain process that demands a new architectural paradigm: **cryptographic agility**.

Defining Cryptographic Agility

Cryptographic agility (or "crypto-agility") is a concept that extends beyond merely supporting multiple algorithms. NIST defines it as the capacity for a system to be rapidly updated to support new cryptographic algorithms, standards, and parameters, ideally without requiring significant changes to the system's core infrastructure [20].

In a truly agile system, the specific cryptographic mechanisms (such as the key exchange algorithm, the signature scheme, or the symmetric cipher) are not deeply embedded in the application logic. Instead, they are treated as replaceable modules or as parameters that can be defined and enforced by an external policy. This separation of "mechanism" from "policy" is the cornerstone of agility and allows an organisation to:

- React quickly to the discovery of a new vulnerability in a deployed algorithm.
- Adopt new, more efficient, or more secure standards as they become available.
- Enforce different cryptographic policies for different users, data types, or communication partners.

The Rationale: Why Agility is Non-Negotiable in the PQC Era

In the context of the PQC migration, this abstract concept becomes a concrete and non-negotiable requirement. The landscape is too new, and the stakes are too high, to risk being "locked in" to a single set of algorithms. The rationale for this is multi-faceted:

1. **The Volatility of New Standards (The SIKE Precedent):** As discussed in Section 2.1.2, the NIST competition hosted algorithms from several different mathematical families. The isogeny-based family was, for a long time, a promising contender due to its very small key sizes. The leading KEM candidate, SIKE (Supersingular Isogeny Key Encapsulation), was a Round 3 alternate and was being actively researched.

   In August 2022, in an event that shocked the cryptographic community, researchers Wouter Castryck and Thomas Decru published a paper detailing a complete and practical key recovery attack on SIKE [21]. The attack, which used classical computer techniques, was devastatingly efficient and effectively broke the entire scheme.

   This event serves as a powerful and practical reminder: these algorithms, while scrutinised, are based on mathematical assumptions that are far less battle-tested than RSA or ECC. A similar flaw could, in theory, be discovered in any of the new standards. A system that has "hard-coded" an algorithm found to be vulnerable would be catastrophically compromised. An agile system, by contrast, could disable the broken algorithm via a simple policy change and transition to an alternative (like the hash-based SLH-DSA) immediately.

2. **The Threat of Implementation Attacks (Side-Channels):** Beyond the mathematical foundations, agility is also an active defence against *implementation* flaws. SCA exploit information leaked from a system's physical implementation, such as power consumption, electromagnetic radiation, or timing, rather than attacking the mathematics [22].

   The new lattice-based standards, ML-KEM and ML-DSA, are particularly susceptible if not implemented with extreme care. For example, timing variations in memory access ("cache-timing attacks") can leak information about the secret key [22], and the "rejection sampling" step in ML-DSA can also leak secret data if not implemented in constant time [23].

   This threat model is different from SIKE: the algorithm's mathematics remains secure, but a specific implementation (like a version of `liboqs`) could be vulnerable. In this scenario, a static, "hard-coded" gateway would be completely compromised until it could be patched, recompiled, and redeployed. A crypto-agile architecture, like the one proposed in this thesis, can react immediately at the policy level. An administrator could disable the compromised algorithm (e.g., 'mlkem768') and enforce a switch to a non-vulnerable alternative (like the hash-based SLH-DSA) via a simple policy update, providing an immediate mitigation without service interruption. Agility is therefore not just a migration strategy, but a critical, real-time defence mechanism.

3. **The Prolonged Transitional Period (Hybridisation):** (Il tuo testo originale, che prima era il punto 2) The global migration to PQC will take years, if not decades. During this time, systems will exist in a "hybrid" state, needing to communicate with both modern, PQC-capable peers and legacy, classical-only peers [24].

   Furthermore, to hedge against the risk of unknown flaws in new algorithms (like the SIKE break or SCA), many protocols are adopting a hybrid key exchange model. In this model, a shared secret is established by combining the output of a classical KEM (like ECC) and a PQC KEM (like ML-KEM). The resulting key is secure as long as **at least one** of the two algorithms remains unbroken.

   This hybrid, transitional environment demands agility. A gateway must be able to negotiate and enforce policies like "Must use PQC if available", "Allow classical-only for legacy client X", or "Enforce PQC-ECC hybrid for all traffic".

4. **Performance and Contextual Trade-offs:** (Il tuo testo originale, che prima era il punto 3) The new PQC standards are not one-size-fits-all. They present a wide range of trade-offs:

   - **ML-KEM (Kyber)** is very fast and has small keys, making it ideal for general-purpose protocols like IKEv2.
   - **ML-DSA (Dilithium)** is also fast, with reasonably small signatures.
   - **SLH-DSA (SPHINCS+)** is much slower and has significantly larger signatures (approx. 8-49 KB, depending on the level) but is based on the robust security of hash functions.

   An agile system can make intelligent, policy-based decisions based on context. A high-bandwidth, low-latency VPN server (like our gateway) might be required to use ML-KEM. An IoT device with extreme bandwidth constraints, however, might be forbidden from using SLH-DSA due to the signature size. A high-assurance system might be required to **only** use SLH-DSA due to its conservative security. An agile architecture is required to manage this complexity.

5. **Evolving Compliance and Policy Requirements:** (Il tuo testo originale, che prima era il punto 4) Finally, the policy landscape will evolve. A government or corporation might initially mandate NIST Level 1, but as quantum computers become more powerful, they may later mandate a shift to Level 3 or Level 5. An agile system allows these compliance changes to be implemented as administrative policy updates, rather than as costly and disruptive software engineering projects.

In the end, the PQC transition is defined by uncertainty, volatility, and a need for gradual, hybrid adoption. These factors make cryptographic agility the central architectural requirement

for any long-lived, secure system. This thesis directly addresses this need by designing a gateway where the cryptographic "what" (the algorithm) is decoupled from the operational "how" (the protocol logic) and is instead governed by an external, easily modifiable policy.

2.2 Securing Networks with IPsec and IKEv2

Having established **why** the PQC migration is necessary, this section analyses **where** this migration must take place. Our work focuses on the protection of VPNs, which are predominantly implemented using the IPsec protocol suite. Unlike application-layer protocols such as TLS, IPsec operates at Layer 3 (the Network Layer) of the ISO/OSI model. This characteristic makes it transparent to applications, but also a fundamental and sensitive component of the infrastructure.

The IPsec architecture, defined in [25], is not a single protocol but a suite that provides a comprehensive framework for security at the IP layer. It logically divides its functions into a "data plane" (how data is protected) and a "control plane" (how the keys and policies to protect that data are negotiated).

2.2.1 The IPsec Protocol Suite: Data Plane

The IPsec data plane defines the packet formats used to protect traffic. Two primary protocols fulfill this role: the Authentication Header (AH) and the Encapsulating Security Payload (ESP).

AH: As defined in [26], AH provides data integrity, data origin authentication, and anti-replay protection. It does this by adding a new header to the IP packet. It is important to note that AH **does not provide confidentiality**; the packet payload remains in clear text. Due to this limitation and its well-known incompatibility with Network Address Translation (NAT), its use is rare in modern VPN implementations.

ESP: ESP, defined in [27], is the workhorse of modern VPNs. ESP provides all the services of AH (integrity, authentication, anti-replay) and, critically, adds **confidentiality** through encryption of the IP packet's payload. This is the protocol we will focus on in this thesis.

Transport Mode vs. Tunnel Mode

Both AH and ESP can operate in two distinct modes, which define **which part** of the original packet is protected [25].

- **Transport Mode:** In this mode, only the payload of the original IP packet is protected (encrypted and/or authenticated). The original IP header is retained. This mode is typically used for host-to-host communications, where the two endpoints of the communication are also the cryptographic endpoints.
- **Tunnel Mode:** In this mode, the entire original IP packet (both header and payload) is protected and encapsulated within a **new** IP packet. This mode is the basis for "gateway-to-gateway" or "client-to-gateway" VPNs. It allows a gateway to act as a cryptographic "proxy" for an entire network, hiding the internal IP addresses of the protected network. **This thesis focuses exclusively on Tunnel Mode**, as we are building a security gateway.

Security Associations and Policy Management

The ESP and AH protocols define **what** a protected packet looks like, but they do not state **which** algorithms or keys to use. This information is held in a logical construct called a **Security Association (SA)** [25].

An SA is a simplex (one-way) connection that defines the parameters for secure communication. For a typical bidirectional ESP connection, two SAs are required (one inbound, one outbound). Each SA contains, amongst other things:

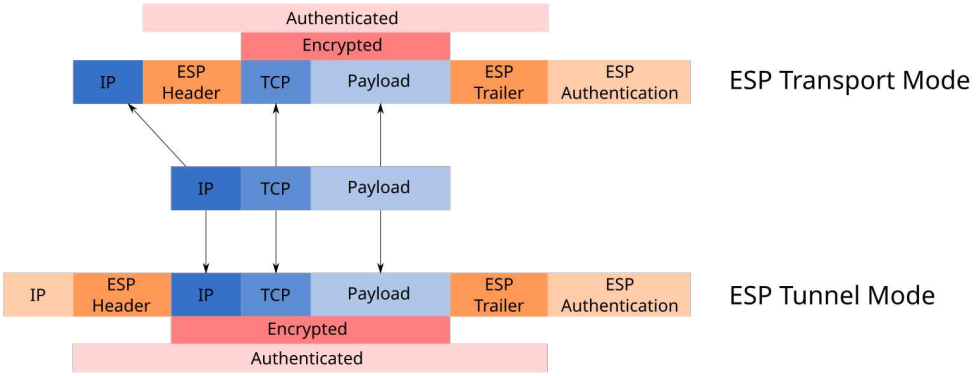

Figure 2.1. Conceptual illustration of IPsec Transport Mode vs. Tunnel Mode.

- The encryption algorithm (e.g., AES-256).
- The integrity algorithm (e.g., SHA-256).
- The cryptographic keys for those algorithms.
- A sequence number (for anti-replay protection).
- The SA "lifetime" (how long it can be used before renegotiation).

An IPsec host manages two fundamental databases for its operation:

Security Policy Database (SPD): This database, managed by the administrator, defines **which** traffic must be protected. It is the "decision maker". For example, a policy in the SPD might state: "All traffic from network A to network B must be protected with ESP in Tunnel Mode".

Security Association Database (SAD): This database contains the parameters of all active SAs. It is the "mechanism" that the kernel uses to process packets.

When an outbound packet matches a rule in the SPD and no valid SA yet exists for that flow, the system determines that a new SA is required. This triggers the IPsec "control plane".

The Role of IKE (Internet Key Exchange)

The process of manually creating SAs (by setting keys by hand) is complex, unscalable, and insecure. For this reason, a control plane protocol was created to automate this negotiation: the **Internet Key Exchange (IKE)**.

IKE is the protocol that two IPsec gateways use to:

1. Authenticate each other (using digital certificates or pre-shared keys).
2. Negotiate the algorithms to be used (e.g., "I support Kyber and AES, what do you support?").
3. Perform a key exchange (like Diffie-Hellman, or, in our case, a PQC KEM) to securely generate session keys.
4. Create the SAs and populate them into the SAD on both sides.

Without IKE, IPsec cannot function automatically. It is here, within the IKE protocol, that the vulnerability to Shor's algorithm resides, and it is here that our PQC solution must intervene. The next section will analyse the modern version of this protocol, IKEv2, in detail.

2.2.2 IKEv2 Protocol (RFC 7296)

As introduced in Section 2.2.1, the IKE protocol is the control plane responsible for peer authentication and the automatic negotiation of SAs. The modern and predominant version of this protocol is IKEv2, defined in [17].

IKEv2 was designed to overcome the significant complexities and ambiguities of its predecessor, Internet Key Exchange version 1 (IKEv1) [28]. Whilst IKEv1 had two complex "Phases" with multiple negotiation modes (Main Mode, Aggressive Mode, Quick Mode), IKEv2 dramatically simplifies this process. The entire IKEv2 negotiation consists of just two exchanges (totalling four messages): the `IKESA_INIT` and the `IKE_AUTH` exchanges.

The goal of IKEv2 is to establish a secure control channel, called the **IKE Security Association (IKESA)**, and subsequently use that channel to negotiate one or more **CHILDSAs**, which correspond to the IPsec SAs (defined in the SAD) used to protect data traffic (the "data plane").

The IKESA_INIT Exchange

The first exchange, `IKESA_INIT`, is intended to establish a secure IKESA. At this stage, the peers are not yet authenticated, but they negotiate cryptographic parameters and generate a shared secret. This exchange consists of two messages:

Message 1 (Initiator → Responder): Contains a header (`HDR`), the Initiator's cryptographic proposal (`SAi1`), its key exchange contribution (`KEi`), and a nonce (`Ni`).

The `SAi1` payload is fundamental: it is a proposal listing the algorithms the Initiator is willing to use for the IKESA. It contains:

- Encryption Algorithm (e.g., AES-CBC).
- Integrity Algorithm (e.g., SHA-256).
- Pseudo-Random Function (Pseudo-Random Function (PRF)) (e.g., HMAC-SHA-256).
- **Diffie-Hellman (Diffie-Hellman (DH)) Group** (e.g., Group 14, a 2048-bit MODP [29]).

The `KEi` payload contains the Initiator's public DH value for the proposed group(s).

Message 2 (Responder → Initiator): Contains the header (`HDR`), the `SAr1` payload (which selects one of the Initiator's proposals), the Responder's DH contribution (`KEr`), and its nonce (`Nr`). It may also contain a `CERTREQ` if the Responder requires a certificate for authentication.

At the conclusion of this exchange, both peers use the DH contributions (`KEi`, `KEr`) to calculate a shared secret. This secret, combined with the nonces (`Ni`, `Nr`), is passed through the negotiated PRF to generate a master secret key called `SKEYSEED`.

From `SKEYSEED`, all necessary cryptographic keying material (Keying Material (SK)) for the session is derived:

- `SK_d`: Keying material used to derive all future CHILDSAs.
- `SK_e/a`: Encryption and authentication keys to protect subsequent IKEv2 messages.
- `SK_p`: Keys used to generate the authentication (`AUTH`) payloads.

It is crucial to note that at the end of `IKESA_INIT`, the peers share a secret, but they **have not yet authenticated each other**. The entire exchange is vulnerable to a Man-in-the-Middle (MITM) attack.

The IKE_AUTH Exchange

The second exchange, `IKE_AUTH`, uses the secure channel established by `IKESA_INIT` (using the `SK_e/a` keys) to authenticate the peers and negotiate the first CHILDSA. This exchange also consists of two messages, which are now **fully encrypted**:

Message 3 (Initiator → Responder): Contains `HDR(SK_e)` (encrypted header), and an encrypted payload `SK { IDi, [CERT], [CERTREQ], AUTH, SAi2, TSi, TSr }`.

Message 4 (Responder → Initiator): Contains `HDR(SK_e)` and an encrypted payload `SK { IDr, [CERT], AUTH, SAr2, TSi, TSr }`.

Let us analyse the critical payloads of this exchange:

- `IDi / IDr:` The identity payloads (e.g., an FQDN, an email address) that declare who the peers are. Their encryption is a key advantage of IKEv2 over IKEv1, as it protects identities from eavesdropping.
- `CERT / CERTREQ:` If using public-key authentication, these payloads carry the X.509 certificates.
- `AUTH:` This is the payload that **performs authentication**. It is typically a digital signature (e.g., RSA or ECDSA) computed over a block of data that includes the `IKESA_INIT` messages, the nonces, and the other peer's identity. This payload binds the signer's identity to the DH exchange that occurred earlier, thereby defeating the MitM attack.
- `SAi2 / SAr2:` The proposal and response for the **first CHILDSA**. This is where the algorithms for ESP (e.g., AES-GCM) are negotiated.
- `TSi / TSr:` The Traffic Selectors. These payloads define *which* traffic (as defined in the SPD) must be protected by this CHILDSA (e.g., "all traffic from 192.168.1.0/24 to 10.0.0.0/16").

At the end of this exchange, the IKESA is authenticated, and the first CHILDSA is ready to protect traffic.

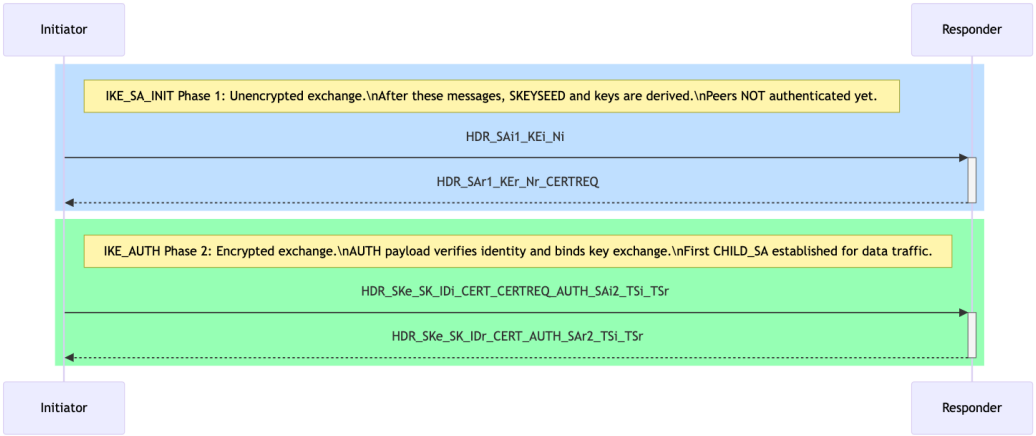

Figure 2.2. Sequence diagram of the IKESA_INIT and IKE_AUTH exchanges.

Summary of PQC Vulnerabilities in IKEv2

This analysis of IKEv2 (RFC 7296) reveals two critical points of failure in the face of a quantum computer running Shor's algorithm (as discussed in Section 2.1.1):

1. **Confidentiality (Key Exchange):** The `IKESA_INIT` exchange bases its security on the DH (or ECC) key exchange. An adversary who captures this exchange can use Shor's algorithm to retroactively compute the shared secret, decrypt the entire `IKE_AUTH` negotiation, and recover all CHILDSA keys. This compromises the entire VPN.
2. **Authentication (Signature):** The `AUTH` payload in the `IKE_AUTH` exchange relies on RSA or ECDSA digital signatures. A quantum adversary can use Shor's algorithm to break the gateway's public key (obtained from the `CERT` payload) and compute its private key. This allows the adversary to impersonate the gateway and launch a MitM attack, successfully authenticating itself.

Therefore, to migrate IPsec to a PQC standard, it is not sufficient to address only one of these problems. It is mandatory to replace **both** the DH key exchange **and** the digital signature algorithm with quantum-resistant alternatives. The following RFCs, which are central to this thesis, define exactly how to achieve this.

2.2.3 Integrating PQC into IKEv2

Addressing the quantum vulnerabilities within IKEv2 (Section 2.2.2) necessitates more than a simple replacement of algorithms; it requires careful architectural integration into the protocol's negotiation logic to handle new cryptographic paradigms like hybridity and the potential for future algorithmic evolution. The IETF has developed a layered approach, documented across several key RFCs. RFC 9242 provides the essential mechanisms for integrating PQC KEMs and signatures directly into the standard `IKESA_INIT` and `IKE_AUTH` exchanges [30]. Building upon this, RFC 9243 introduces the generic `IKE_INTERMEDIATE` exchange, offering a flexible framework for adding cryptographic operations [31]. Crucially, RFC 9370 (specifically in its Appendix A) provides concrete guidance on leveraging this intermediate exchange for incorporating **Additional Key Exchanges (ADDKEs)** using PQC KEMs, alongside validating the suitability of PQC-derived keys for the ESP data plane [32]. Together, these documents provide a comprehensive and standardised foundation for building resilient, quantum-resistant IPsec tunnels.

RFC 9242: Direct PQC Integration in Initial Exchanges

RFC 9242 [30] lays the groundwork by extending the existing IKEv2 structure. It introduces new Internet Assigned Numbers Authority (IANA)-registered **Transform Types** (Type 6 for PQC KEMs, Type 7 for PQC Signatures) and specifies how existing payloads (`KE`, `AUTH`) are adapted to carry PQC-specific data.

Post-Quantum Key Encapsulation Mechanisms (KEMs): To replace the vulnerable classical key exchange, RFC 9242 introduces Transform Type 6, specifically for "Post-Quantum Secure Key Exchange Methods," distinct from the classical Type 4 (DH Group) [33].

- **Negotiation Details:** Within the Security Association (`SA`) payload (`SAi1`/`SAr1`) of the `IKESA_INIT` exchange, peers propose cryptographic suites that include transforms of Type 6. Each transform specifies a particular PQC KEM algorithm, identified by a unique IANA-assigned ID. For instance, the IANA registry includes identifiers for various parameter sets of ML-KEM (Kyber), such as `ML_KEM_512_IPD`, `ML_KEM_768_IPD`, and `ML_KEM_1024_IPD` (where "IPD" signifies "IKE Payload Definition", indicating suitability for direct use in IKE). The Responder selects a mutually supported Type 6 transform from the Initiator's proposal list in `SAi1` and includes it in the `SAr1` payload [33].

- **Payload Adaptation for KEM Data:** The Key Exchange (`KE`) payload, traditionally used for DH public values, is repurposed to transport the PQC KEM-specific data. The exact structure within the `KE` payload is defined per KEM algorithm, but for an IND-CCA2 secure KEM like ML-KEM (Kyber), the typical flow as detailed in RFC 9242 is:

  1. The Initiator generates a PQC key pair (pk_i, sk_i) for the negotiated KEM. It sends its public key (pk_i) in the `KEi` payload.
  2. The Responder receives pk_i. It then generates a random shared secret ss_R and encapsulates it using pk_i via the KEM's encapsulation function (Encaps(pk_i)), yielding the shared secret ss_R and a ciphertext ct. The Responder securely stores ss_R and sends ct back in the `KEr` payload.
  3. The Initiator receives ct. It uses its private key sk_i and the KEM's decapsulation function (Decaps(sk_i, ct)) to recover the **same** shared secret ss_I, where $ss_I = ss_R$. If decapsulation fails, the exchange terminates.

  This sequence establishes a shared secret known only to the two peers, forming the basis for the IKESA's cryptographic strength.

- **Integration with Key Derivation Function (KDF):** The shared secret ss (from the PQC KEM decapsulation) directly replaces the classical DH shared secret as a source of entropy. This ss, combined with the nonces (`Ni`, `Nr`), is fed into the negotiated PRF to compute the master secret `SKEYSEED`, following the standard IKEv2 KDF defined in RFC 7296 [17, 33]. From `SKEYSEED` onwards, the derivation of all subsequent keying material (`SK_d`, `SK_e`, `SK_a`, `SK_p`) proceeds without modification, showcasing a seamless integration into the existing key hierarchy.

Post-Quantum Digital Signatures: The second critical vulnerability is the reliance on classical digital signatures (RSA, ECDSA) within the `AUTH` payload of the `IKE_AUTH` exchange. RFC 9242 addresses this by introducing Transform Type 7 for "Post-Quantum Secure Authentication Methods" [33].

- **Negotiation and Identifiers:** While the specific choice of signature algorithm is often tied to the certificate presented and the "Authentication Method" field in the `SA` payload, Transform Type 7 provides formal IANA identifiers for PQC signature schemes (e.g., for various variants of ML-DSA and SLH-DSA). This allows peers to explicitly signal their capabilities and preferences, crucial for cryptographic agility [33].

- **AUTH Payload Calculation:** The process for calculating the `AUTH` payload remains consistent with RFC 7296 [17]. A canonical data block, comprising elements from the `IKESA_INIT` messages, the peer's nonce, and the signer's own identity payload (`IDi` or `IDr`), is constructed. The signer then computes a digital signature over this data block using their PQC private key, which corresponds to the negotiated PQC signature algorithm. The resulting signature value is then placed in the `AUTH` payload [33].

- **Verification and PQC Certificates:** The receiver verifies the signature using the sender's PQC public key, typically obtained from an X.509 certificate conveyed in the `CERT` payload during the `IKE_AUTH` exchange. RFC 9242 anticipates that PQC public keys and signatures can be significantly larger than their classical counterparts. This necessitates new certificate formats or extensions (e.g., using specific OIDs for PQC public keys and signature algorithms within X.509 structures, a topic addressed by parallel PKI standardisation efforts). Regardless of the certificate format, successful verification confirms a strong cryptographic binding: the entity possessing the private key corresponding to the certified public key did indeed participate in and endorse the `IKESA_INIT` exchange, thereby effectively thwarting MitM attacks [33].

Formalising and Detailing Hybrid Key Exchange: The dynamic and uncertain nature of the PQC transition (as discussed in Section 2.1.5) makes **hybrid key exchange** a cornerstone of robust security. RFC 9242 provides specific mechanisms for combining classical (DH/ECC) and PQC KEMs during the `IKESA_INIT` exchange.

- **Negotiation of Multiple Key Exchange Methods:** To establish a hybrid IKESA, the Initiator proposes **multiple** key exchange transforms within its `SAi1` payload. This includes both a classical transform (e.g., Type 4, specifying a well-known ECC group like P-384) and a PQC KEM transform (Type 6, specifying e.g., `ML_KEM_768_IPD`). The Responder indicates agreement by returning the selected classical and PQC transforms in its `SAr1` payload [33].
- **Transporting Multiple KE Payloads:** The `IKESA_INIT` messages are designed to carry multiple `KE` payloads. RFC 9242 specifies that these payloads are ordered consistently with the negotiation. For a hybrid exchange, `KEi` would contain the Initiator's classical public value followed by its PQC KEM public key, and `KEr` would contain the Responder's corresponding contributions [33].
- **Combining Shared Secrets (The Concatenation Method):** Once both the classical and PQC key exchanges are successfully completed, yielding two distinct shared secrets ($ss_{classical}$ and ss_{pqc}), these are combined. RFC9242 [33] mandates a simple, cryptographically sound concatenation: $ss_{combined} = ss_{classical}||ss_{pqc}$ (where $||$ denotes bit string concatenation). This $ss_{combined}$ then serves as the input to the negotiated PRF for deriving `SKEYSEED` [33]. This technique ensures that the final keying material benefits from the entropy of both algorithms.

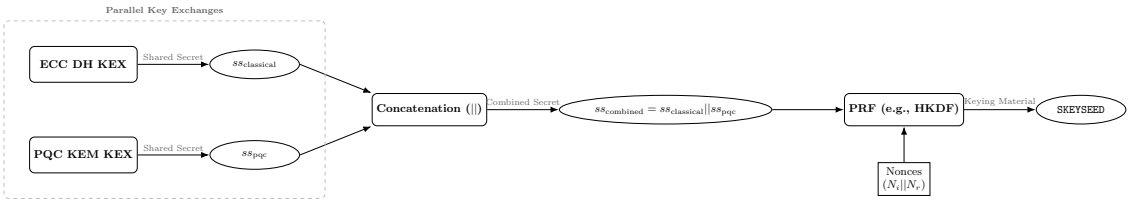

Figure 2.3. Conceptual illustration of Hybrid Key Exchange secret combination.The shared secrets from both classical (DH/ECC) and PQC (KEM) exchanges are concatenated before being fed into the PRF (Pseudo-Random Function) for `SKEYSEED` derivation, along with nonces ($N_i||N_r$).

- **Security Rationale for Hybridisation:** The profound security advantage of this hybrid approach is its resilience against evolving threats. The resulting `SKEYSEED` (and all subsequent derived keys) remains secure as long as *at least one* of the component key exchange mechanisms resists attack. This strategy offers "multi-factor security": it protects against quantum adversaries (assuming the PQC KEM is secure) while simultaneously providing continued security against classical cryptanalysis even if the PQC KEM were to be unexpectedly broken by classical means. This "defense-in-depth" is critical during the uncertain transition period to PQC [33, 24]. Our gateway implementation explicitly leverages this hybrid capability to provide robust, forward-looking security.

In essence, RFC 9242 meticulously details how PQC algorithms can be woven into the existing IKEv2 fabric, enabling both direct replacements and robust hybrid solutions. However, the IETF recognized that sometimes a more dynamic approach for adding cryptographic strength or algorithms **after** an initial secure channel is established might be necessary.

RFC 9243 and RFC 9370 (Appendix A): Intermediate Exchanges for Additional PQC KEMs

While RFC 9242 provides a complete method for establishing a PQC-secured or hybrid IKESA from the outset, it doesn't cover all possible scenarios for incorporating quantum-resistant cryptography, especially when additional algorithmic layers or dynamic updates are desired. This

is where RFC 9243 [34] introduces a pivotal concept: the **IKEv2 Intermediate Exchange** (`IKE_INTERMEDIATE`). This exchange type allows for a flexible insertion of additional messages after the initial `IKESA_INIT` but before the final `IKE_AUTH` exchange is completed, or even later during rekeying.

Crucially, **RFC 9370, particularly in Appendix A**, provides concrete guidance and detailed examples demonstrating how this generic `IKE_INTERMEDIATE` exchange can be leveraged specifically for performing **ADDKEs** using PQC KEMs [35]. This addresses scenarios beyond the initial hybrid setup defined in RFC 9242, enabling layered and iterative security enhancements.

Purpose and Flexibility of IKE_INTERMEDIATE for PQC: The primary purpose of `IKE_INTERMEDIATE` (as enabled for PQC by RFC 9370 Appendix A) is to allow peers to perform **Additional Authenticated Key Exchanges (AAKEs)** or other cryptographic computations that dynamically extend the IKESA's security or capabilities [34, 35]. This is particularly relevant for PQC in several scenarios:

- **Adding PQC to Existing Classical IKESA:** If an IKESA was initially established with only classical cryptography, an `IKE_INTERMEDIATE` exchange could be used to perform a separate PQC KEM and mix its shared secret into the existing IKESA keying material, effectively "upgrading" its security to hybrid without tearing down the entire session.
- **Separating PQC KEM from Authentication:** In some architectures, it might be desirable to perform a (potentially large and slow) PQC KEM in a separate exchange before the final `AUTH` step. This can improve the responsiveness of the initial `IKE_AUTH` phase or provide more granular control over cryptographic elements, especially when dealing with the latency associated with some PQC primitives.
- **Multiple PQC KEMs / Layering:** For very high-security requirements or to leverage cryptographic diversity (e.g., combining lattice-based and code-based PQC KEMs), an `IKE_INTERMEDIATE` exchange could allow for the negotiation and execution of **multiple** distinct PQC KEMs (even from different cryptographic families) to further enhance quantum resistance through diversity and provide a greater "security budget".
- **Post-Quantum Rekeying:** When the IKESA or CHILDSAs are rekeyed (e.g., for Perfect Forward Secrecy or to refresh cryptographic material), `IKE_INTERMEDIATE` can be used to perform a fresh PQC KEM to generate new quantum-resistant keys. This is crucial for protecting against future quantum attacks even if previous traffic was harvested and later decrypted.

Negotiating Additional Key Exchanges (ADDKE): Support for these additional exchanges is negotiated upfront during the `IKESA_INIT` phase, extending the 'SA' payload negotiation, as detailed in RFC 9370 Appendix A [35]:

- **New Transform Types for ADDKEs:** Specific Transform Types are defined for signalling additional key exchanges (e.g., `ADDKE1`, `ADDKE2`, etc.). These are included in the `SAi1`/`SAr1` payloads alongside the primary key exchange (`KE`) transform.
- **Proposing Options including NONE:** Within each `ADDKE` transform type, the Initiator proposes one or more specific PQC KEM algorithm identifiers (e.g., proposing `PQ_KEM_1` or `PQ_KEM_2` for `ADDKE1`). Critically, the proposal for each `ADDKE` type **must** also explicitly include `NONE` as a mandatory option. This mechanism allows the Responder to gracefully decline any specific additional exchange without failing the entire `SA` negotiation [35].
- **Responder Selection**: For each `ADDKE` type proposed, the Responder must select exactly one algorithm identifier (which can be `NONE`) and include these selections in the `SAr1` payload [35].
- **Signalling General Support for Intermediate Exchanges:** Beyond specific `ADDKE` negotiation, both peers must also signal their general capability to handle intermediate exchanges by including the `INTERMEDIATE_EXCHANGE_SUPPORTED` notification payload in their respective `IKESA_INIT` messages [34].

Performing the Additional Key Exchange(s) via IKE_INTERMEDIATE: For every ADDKE transform successfully negotiated with a specific PQC KEM (i.e., not NONE), the peers execute a dedicated IKE_INTERMEDIATE exchange. These exchanges occur sequentially **after** IKESA_INIT is complete and **before** the IKE_AUTH exchange begins [35].

- Each IKE_INTERMEDIATE exchange comprises a request/response pair and is fully protected (encrypted and authenticated) using the keys (SK_e/SK_a) derived from the currently established keying material (initially from the IKESA_INIT, and subsequently from any prior ADDKE updates) [34].
- The purpose of each exchange is to perform one specific ADDKE. The messages carry the PQC KEM public key (e.g., in KEi(n)) and the resulting ciphertext (e.g., in KEr(n)), where 'n' uniquely identifies which additional exchange is being performed (e.g., KEi(1) for ADDKE1, KEi(2) for ADDKE2, etc.) [35].
- Upon successful completion of an IKE_INTERMEDIATE exchange for an ADDKE, a new shared secret ($SK(n)$) is obtained from that specific PQC KEM.

Iterative Keying Material Update (SKEYSEED): A core aspect detailed in RFC 9370 Appendix A is how the entropy from these additional exchanges is iteratively incorporated into the master secret SKEYSEED [35]. Each new shared secret $SK(n)$ derived via an IKE_INTERMEDIATE exchange is immediately used to **update and strengthen** the current SKEYSEED through a well-defined KDF procedure:

1. Let SKEYSEED(0) be the master secret derived from the initial IKESA_INIT exchange (which might itself be hybrid per RFC 9242). Let SK_d(0) be the corresponding key derivation key.
2. After the first ADDKE yields secret $SK(1)$:
   `SKEYSEED(1) = prf(SK_d(0), SK(1) | Ni | Nr)`
3. Derive new keys from SKEYSEED(1):
   `{SK_d(1) | SK_a*(1) | SK_ar(1) | SK_ei(1) | SK_er(1) | SK_pi(1) | SK_pr(1)} = prf+(SKEYSEED(1), Ni | Nr | SPIi | SPIr)`
4. After the second ADDKE yields secret $SK(2)$:
   `SKEYSEED(2) = prf(SK_d(1), SK(2) | Ni | Nr)`
   `{SK_d(2) | SK_a*(2) | SK_ar(2) | SK_ei(2) | SK_er(2) | SK_pi(2) | SK_pr(2)} = prf+(SKEYSEED(2), Ni | Nr | SPIi | SPIr)`

This process repeats for every completed ADDKE. The keys derived after the **final** update (SK_p*(last), SK_e*(last), etc.) are those used for the subsequent IKE_AUTH exchange and for deriving CHILDSA keys [35]. This ensures that the final IKESA incorporates entropy from the primary key exchange **and** all successfully negotiated additional PQC key exchanges, significantly enhancing resilience.

Example Exchange Flow (Based on RFC 9370 Appendix A.1): Consider the scenario from [35] where the Initiator proposes a primary classical key exchange (KE=Curve25519) and three optional additional PQC KEM exchanges (ADDKE1, ADDKE2, ADDKE3). The negotiation and subsequent exchanges proceed as follows:

```
Initiator ------------------------------------------------ Responder
-----------------------------------------------------------------
HDR(IKESA_INIT), SAi1(KE=Curve25519,
                      ADDKE1=PQ_KEM_1|PQ_KEM_2|NONE,
                      ADDKE2=PQ_KEM_3|PQ_KEM_4|NONE,
                      ADDKE3=PQ_KEM_5|PQ_KEM_6|NONE, ...),
KEi(Curve25519), Ni, N(INTERMEDIATE_EXCHANGE_SUPPORTED) --->
```

```
                    <--- HDR(IKESA_INIT), SAr1(KE=Curve25519,
                                               ADDKE1=PQ_KEM_2,
                                               ADDKE2=NONE,
                                               ADDKE3=PQ_KEM_5, ...),
                                KEr(Curve25519), Nr, N(
                                    INTERMEDIATE_EXCHANGE_SUPPORTED)

<-- Peers calculate SKEYSEED(0) based on Curve25519 shared secret -->
<-- Peers derive initial SK_d(0), SK_e(0), SK_a(0), SK_p(0) -->

HDR(IKE_INTERMEDIATE), SK {KEi(1)(PQ_KEM_2)} ---> [Protected with SK_e(0)/SK_a
    (0)]
                    <--- HDR(IKE_INTERMEDIATE), SK {KEr(1)(PQ_KEM_2)}

<-- Peers obtain SK(1) from PQ_KEM_2 -->
<-- Peers update SKEYSEED: SKEYSEED(1)=prf(SK_d(0), SK(1)|Ni|Nr) -->
<-- Peers derive updated SK_d(1), SK_e(1), SK_a(1), SK_p(1) -->

HDR(IKE_INTERMEDIATE), SK {KEi(2)(PQ_KEM_5)} ---> [Protected with SK_e(1)/SK_a
    (1)]
                    <--- HDR(IKE_INTERMEDIATE), SK {KEr(2)(PQ_KEM_5)}

<-- Peers obtain SK(2) from PQ_KEM_5 -->
<-- Peers update SKEYSEED: SKEYSEED(2)=prf(SK_d(1), SK(2)|Ni|Nr) -->
<-- Peers derive final SK_d(2), SK_e(2), SK_a(2), SK_p(2) -->

HDR(IKE_AUTH), SK{ IDi, AUTH, SAi2, TSi, TSr } ---> [Protected with SK_e(2)/
    SK_a(2)/SK_p(2)]
                    <--- HDR(IKE_AUTH), SK{ IDr, AUTH, SAr2, TSi, TSr }
```

In this example, the Responder opted out of the second additional exchange (`ADDKE2=NONE`) but agreed to the first (`ADDKE1=PQ_KEM_2`) and third (`ADDKE3=PQ_KEM_5`). Two separate `IKE_INTERMEDIATE` exchanges follow `IKESA_INIT`, each updating the SKEYSEED and derived keys. The final `IKE_AUTH` exchange uses the keys derived from the fully updated `SKEYSEED(2)`. This demonstrates the flexibility and robust, layered security achieved through this mechanism for incorporating multiple PQC key agreements.

RFC 9370 (Main Body): Reconfirming ESP Key Suitability

Finally, the main body of RFC 9370 performs the essential validation regarding the suitability of keys derived through these potentially complex, multi-stage KDF processes for use within the ESP data plane [35]. The concern centres on whether the final derived keys possess sufficient pseudorandomness and quality to meet the stringent demands of modern ESP ciphers, particularly Authenticated Encryption with Associated Data (AEAD) (Authenticated Encryption with Associated Data) algorithms.

AEAD modes, such as AES-GCM [27], are the cornerstone of contemporary ESP security, offering both confidentiality and integrity in a single cryptographic primitive. Their security relies on stringent assumptions about the randomness and uniqueness of the keys and initialization vectors (IVs) used. A KDF that produces statistically weak or predictable keys from its input could undermine these assumptions, even if the underlying AEAD cipher itself is strong.

RFC 9370 specifically investigates the robustness of the IKEv2 Key Derivation Function (KDF), which uses the negotiated PRF (e.g., HMAC-SHA256) in a "prf+" construction, when its initial entropy source (`SKEYSEED`) incorporates shared secrets from PQC KEMs (via RFC 9242) or from the iterative updates involving multiple PQC secrets (via the `ADDKE` mechanism described in RFC 9370 Appendix A) [35]. The "prf+" construction is defined as an iterative application of

the PRF to expand and mix entropy from the (potentially updated) `SKEYSEED` with other context (e.g., nonces, SPI values) to generate arbitrary amounts of keying material for CHILDSAs. For example: Key Material = prf($SK\_d, N|S|\text{SPI}|\ldots$) where SK_d is derived from `SKEYSEED`.

The analysis in RFC 9370 arrives at key conclusions regarding this process [35]:

- **PRF Robustness:** The security of the derived keys fundamentally depends on the strength of the underlying PRF. As standard PRFs like HMAC-SHA256 are believed to be quantum-resistant (with their security level only quadratically reduced by Grover's algorithm, a manageable effect), the KDF itself remains robust in the quantum era.
- **Entropy Mixing and Expansion:** The iterative "prf+" process is highly effective at mixing and expanding the initial entropy from the (potentially iteratively updated) `SKEYSEED`. Even if a raw shared secret from a PQC KEM were to exhibit subtly different statistical properties compared to a classical DH secret, the cryptographic strength of the PRF effectively randomises the output. This ensures that the derived keys are computationally indistinguishable from random for use in AEAD ciphers.
- **No KDF Modifications Needed:** Crucially, RFC 9370 concludes that no modifications are required for the standard IKEv2 key derivation procedures when generating ESP keys from PQC KEMs (or hybrid combinations, or multiple `ADDKE` updates), as long as these are integrated according to RFC 9242, RFC 9243, and the guidelines in RFC 9370 Appendix A.

In essence, RFC 9370 provides the critical validation that the secure key establishment provided by RFC 9242 (direct integration) and RFC 9243/9370 Appendix A (flexible additions) seamlessly translates into robust keying material for the ESP data plane. This completes the end-to-end security picture, confirming that IPsec can be fully quantum-resistant from initial key negotiation to data encryption, accommodating various levels of cryptographic layering.

With these standardised mechanisms (direct integration, flexible additions via intermediate exchanges, and validated key derivation) providing a comprehensive and versatile toolkit, the focus shifts to the practical software components and architectural patterns available for building such systems. We begin this exploration by discussing the specific IPsec implementation chosen as the foundation for this thesis: strongSwan.

2.3 Enabling Technologies and Architectures

Having established the theoretical underpinnings of PQC and the protocol-level mechanisms for its integration into IPsec via IKEv2, we now turn our attention to the practical software components and architectural patterns required to build a functional PQC-agile gateway. This section provides an overview of the key technologies selected for this thesis work: the strongSwan IPsec implementation, the Open Quantum Safe (OQS) project for providing PQC algorithm implementations, and the Open Policy Agent (OPA) for externalised policy management.

2.3.1 strongSwan: The Chosen IPsec Platform

The choice of a foundational IPsec platform is a critical design decision. In the Linux ecosystem, the primary open-source contenders, strongSwan and Libreswan, both trace their lineage to the FreeS/WAN project [36]. However, their modern architectures diverge significantly. Libreswan, the default in many RHEL-based distributions, is architecturally monolithic in its cryptography, integrating deeply with the **Network Security Services (NSS)** library and delegating all IKE crypto functions to it [37]. While this approach is stable and simplifies FIPS compliance [38], it makes adding new or experimental cryptographic primitives (like PQC) a complex undertaking that would require modifying the core NSS integration.

In sharp contrast, strongSwan was redesigned with a highly extensible, multi-threaded architecture built around a powerful **plugin system** [39, 36]. This modularity is its key strategic

advantage and the principal reason for its selection in this thesis. It allows third-party features to be added as discrete plugins without altering the core daemon. This very flexibility is what the Open Quantum Safe (OQS) project leverages to provide 'liboqs' as a dedicated "oqs" plugin [40], and it is the same mechanism that enables the external authorisation hook central to our policy-based design.

Several other key factors reinforce this choice:

- **Open Source and Transparency:** As an open-source project, strongSwan's codebase is freely available for inspection, modification, and extension [41]. This was vital for understanding its internals, debugging integration, and developing the custom patches and scripts for our policy hook.
- **Standards Compliance and Feature Richness:** The platform has a strong reputation for adhering closely to IETF standards, including IKEv2 [42] and its many extensions.
- **Active Development:** The project is actively maintained with regular updates and a responsive community [41]. This is crucial in the rapidly evolving PQC field, ensuring the platform does not become a security liability.

This combination of an extensible plugin architecture, robust standards compliance, and active development made strongSwan the only viable platform for building a flexible, forward-looking, PQC-agile gateway.

Core Architecture

strongSwan's architecture separates the key negotiation (control plane) from the packet processing (data plane), aligning well with the conceptual model of IPsec itself [43].

IKE Daemon (charon): The central component is the IKE daemon, named 'charon'. This user-space daemon is responsible for handling all IKEv1 and IKEv2 negotiations. It manages the IKESA and CHILDSA states, performs cryptographic operations (key exchange, encryption/decryption of IKE messages, signature generation/verification), interacts with authentication backends (e.g., certificate stores, EAP methods), and crucially, configures the Operating System (OS) kernel's IPsec stack [43].

Kernel IPsec Stack (Data Plane): strongSwan itself does not typically perform the per-packet encryption/decryption of user data (ESP processing). Instead, once 'charon' establishes a CHILDSA, it configures the underlying OS kernel's native IPsec implementation to handle the data plane operations. On Linux, this is typically done via the **XFRM/NETKEY** interface. The kernel is highly optimised for this task, ensuring efficient packet processing. 'charon' installs the negotiated cryptographic keys, algorithms, and traffic selectors (SPD rules) into the kernel, which then automatically intercepts, encrypts/decrypts, and forwards traffic matching those policies [43].

Plugin System: The functionality of the 'charon' daemon is heavily augmented by plugins. These dynamically loaded libraries provide implementations for specific cryptographic algorithms (e.g., AES-GCM, SHA-256, RSA), authentication methods (EAP types, public key infrastructure handling), network interfaces, logging mechanisms, and advanced features. This modularity allows administrators to load only the necessary components and, more importantly, allows developers (including researchers) to add new capabilities by writing custom plugins [43]. The integration of PQC algorithms via the OQS project and the external policy hook used in this thesis are both realised through this plugin mechanism.

In summary, strongSwan provides a robust, standards-compliant, and highly extensible platform for implementing IPsec-based VPNs. Its open-source nature and powerful plugin architecture make it an ideal choice for integrating and experimenting with emerging technologies like Post-Quantum Cryptography and externalised policy control, forming the practical foundation upon which the system described in this thesis is built. The following sections will delve into the specific plugins and external systems that enable the PQC and crypto-agile capabilities of our gateway.

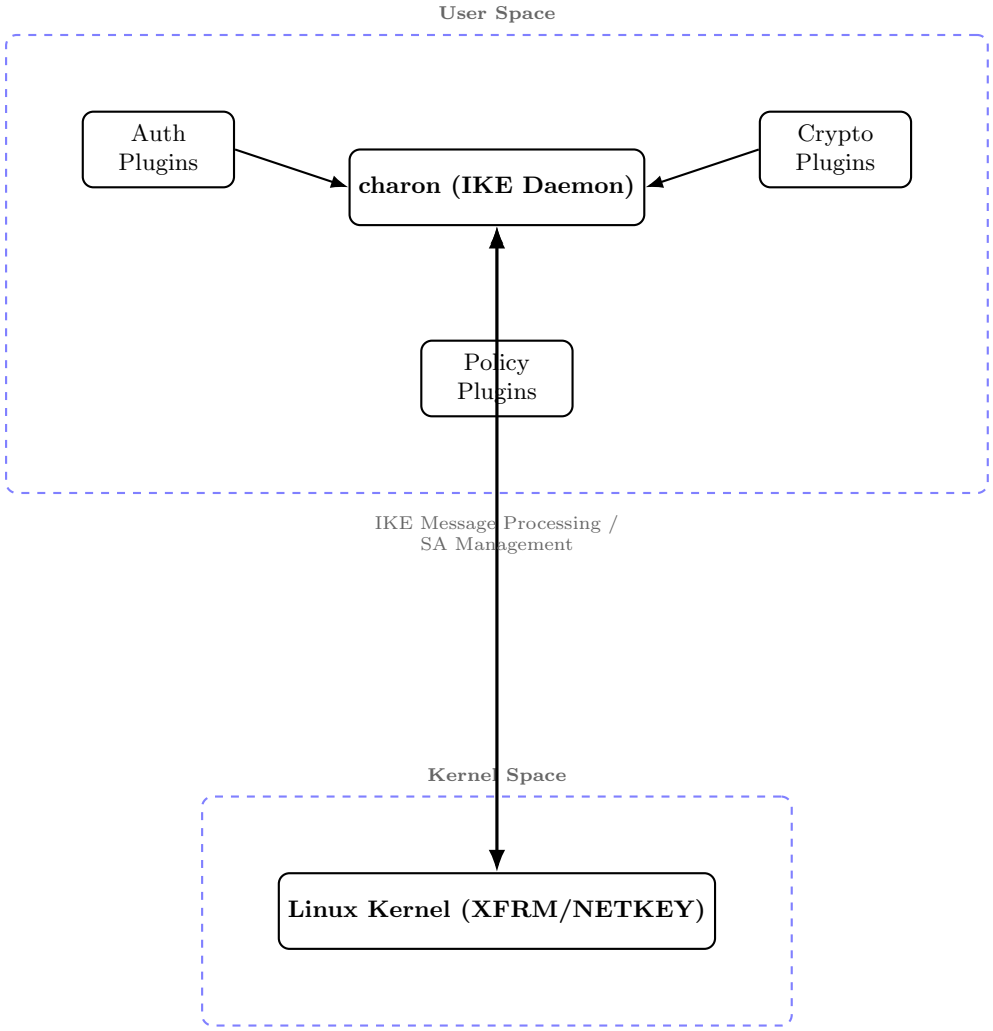

Figure 2.4. Simplified architectural overview of strongSwan, highlighting the user-space IKE daemon (‘charon‘) with its plugin system and its interaction with the kernel’s IPsec data plane.

2.3.2 Open Quantum Safe (OQS): PQC Algorithm Integration

While strongSwan provides the robust and standards-compliant framework for IPsec and IKEv2 negotiations, the platform itself does not inherently contain implementations of the novel PQC algorithms necessary to counter the quantum threat. The complex task of implementing, testing, and optimising these sophisticated cryptographic primitives requires dedicated effort. Integrating such algorithms into existing, complex communication protocols like IKEv2 demands secure, reliable, and well-vetted implementations, preferably accessible through a stable and consistent interface. This crucial role of providing accessible and high-quality PQC implementations is fulfilled admirably by the **Open Quantum Safe (OQS)** project [44].

The Mission and Architecture of the OQS Project: The OQS project is a collaborative, open-source initiative with a clear mission: to support the development and prototyping of quantum-resistant cryptography [44]. It aims to be a central resource for developers and researchers, facilitating the transition towards PQC by offering readily usable implementations of candidate and standardised algorithms. This facilitates experimentation, benchmarking, and ultimately, the integration of PQC into real-world applications and protocols. The project is not developing new cryptographic schemes itself, but rather curating and integrating implementations from the global cryptographic community, particularly those submitted to the NIST standardisation process [45].

OQS achieves its goals through a carefully designed layered architecture:

- **`liboqs`: The Core Cryptographic Library:** At the heart of the project lies `liboqs`, a C library designed for portability and ease of use [45]. Its primary contribution is providing a *unified Application Programming Interface (API)* across a wide spectrum of PQC KEMs and digital signature schemes. This abstraction layer is invaluable; it allows developers to experiment with or switch between different PQC algorithms (e.g., from lattice-based Kyber to hash-based SPHINCS+) with minimal changes to their application code. `liboqs` encapsulates optimised implementations, often incorporating contributions directly from the algorithm designers or leveraging platform-specific optimisations (like AVX2 instructions where available) to enhance performance. It diligently tracks the NIST process, providing implementations not only for the finalised standards (ML-KEM, ML-DSA, SLH-DSA) but also for many candidates from earlier rounds and the ongoing fourth round, making it an essential tool for comparative research and for maintaining crypto-agility [45].
- **Language Wrappers:** Recognising that not all development occurs in C, the OQS project provides wrappers for several popular programming languages (including Python, Java, C#, Go). These wrappers expose the functionality of `liboqs` to a broader developer community, further lowering the barrier to entry for PQC integration.
- **Integration Demonstrators (Forks):** To prove feasibility and provide practical examples, OQS maintains integrated "forks" or patched versions of widely-used network security protocols and libraries. These demonstrators showcase how PQC can be incorporated into existing ecosystems. Notable examples directly relevant to network security include `oqs-OpenSSL` (a modified branch of the ubiquitous OpenSSL library enabling PQC in TLS 1.3) and, crucially for this thesis, `oqs-strongSwan` (a branch of strongSwan demonstrating PQC integration within IKEv2/IPsec) [46]. While these forks serve as vital prototypes and research tools, they often lag behind the main development branches of the upstream projects and require careful management regarding updates and security patches.

Mechanism of OQS Integration within strongSwan: The integration of OQS's capabilities into the strongSwan platform, as demonstrated by the 'oqs-strongSwan' fork and potentially through dedicated plugins in future official releases, is pivotal for constructing a PQC-capable gateway. This integration masterfully leverages strongSwan's modular plugin architecture (detailed in Section 2.3.1) to seamlessly incorporate the suite of PQC algorithms provided by `liboqs` directly into the cryptographic operations managed by the 'charon' IKE daemon.

The primary component facilitating this is typically an **`oqs` plugin** (or a similarly named component). This plugin acts as a crucial bridge:

- **Linking and Registration:** The plugin links against the compiled `liboqs` C library. Upon initialisation, it interacts with strongSwan's cryptographic API to register the PQC algorithms available within `liboqs` (e.g., various parameter sets of ML-KEM, ML-DSA, SLH-DSA, and potentially others) as named cryptographic primitives that 'charon' can understand and negotiate. This registration associates the algorithm names used in strongSwan's configuration (e.g., 'kyber768') with the corresponding functions provided by the `oqs` plugin.
- **Dispatching Operations:** When an IKEv2 negotiation requires a PQC operation, for instance, generating a key pair for ML-KEM, encapsulating a secret, decapsulating a ciphertext during `IKESA_INIT` or `IKE_INTERMEDIATE`, or signing/verifying data for the `AUTH` payload using ML-DSA, the 'charon' daemon identifies the required algorithm as being provided by the `oqs` plugin. It then dispatches the specific cryptographic task (e.g., "encapsulate using this public key") to the plugin.
- **Invoking `liboqs`:** The `oqs` plugin receives the request from 'charon', translates it into the corresponding API call defined by `liboqs`, invokes the appropriate function within the `liboqs` library (which performs the actual PQC computation), and then returns the result (e.g., the ciphertext, the shared secret, the signature, or a verification status) back to 'charon'.

While some highly specialised PQC algorithms might necessitate their own dedicated strongSwan plugins for optimal integration or to handle unique protocol interactions, the general `oqs` plugin provides a versatile and unified interface for the majority of schemes supported by `liboqs`.

This plugin-based integration empowers a 'liboqs'-enabled strongSwan daemon ('charon') with the necessary capabilities to fully support quantum-resistant IKEv2 as defined by the relevant standards:

- It can correctly advertise support for specific PQC KEMs (Type 6 transforms) and signature schemes (Type 7 transforms or via Authentication Method) during the `SA` payload negotiation in `IKESA_INIT`, using the appropriate IANA-assigned identifiers.
- It can execute the required PQC KEM operations (key generation, encapsulation, decapsulation) for both direct integration (RFC 9242) and additional key exchanges via `IKE_INTERMEDIATE` (RFC 9243/9370), handling the specific data formats within the `KE` payloads.
- It can generate and verify PQC digital signatures (e.g., ML-DSA, SLH-DSA) for the `AUTH` payload within the `IKE_AUTH` exchange, ensuring quantum-resistant peer authentication.
- It can seamlessly facilitate hybrid key exchanges by performing both a classical and a PQC key exchange and providing both resulting shared secrets to 'charon''s core KDF logic for concatenation and subsequent `SKEYSEED` derivation, as detailed in Section 2.2.3.

2.3.3 Policy-Based Control Architectures and Engines

The cryptographic agility necessitated by the quantum threat, as discussed in Section 2.1.5, requires more than just support for multiple algorithms within the gateway. It demands an architectural shift towards dynamic, externally manageable security decision-making. Static configurations embedded within the gateway logic are too rigid to adapt quickly to new standards, vulnerabilities, or operational requirements. This leads us to the paradigm of **Policy-Based Network Control (Policy-Based Network Control (PNC))**, specifically focusing on its application to access control and cryptographic negotiation through the PDP/PEP model, and the role of modern policy engines like **Open Policy Agent (OPA)** in realising this model effectively.

The PDP/PEP Model for Access and Cryptographic Control

Policy-Based Network Control moves away from imperative configuration ("do this") towards a declarative approach ("this is the desired state"). At its core lies a well-defined architectural separation of concerns, formalised by the IETF in frameworks like the one for policy-based admission control [47], and often visualised in access control models (see, for example, the XACML reference model discussed in [48]). This separation is primarily embodied by two key functional roles:

- **Policy Enforcement Point (PEP):** This component acts as the gatekeeper, situated directly in the path of execution or communication. In our context, the strongSwan gateway itself functions as the PEP. Its primary responsibilities are to intercept relevant events (e.g., an incoming IKEv2 connection request, specifically the `IKESA_INIT` or `IKE_AUTH` messages containing cryptographic proposals), gather contextual information about the event, request a policy decision from the PDP, and then rigorously enforce the received decision [48]. For instance, if the PDP mandates the use of a specific hybrid PQC suite, the PEP (strongSwan) must ensure that the subsequent negotiation adheres to this mandate, potentially rejecting incompatible proposals or forcing the selection of compliant algorithms. The PEP does not interpret the policies itself; it merely acts upon the authoritative decision it receives [47].
- **Policy Decision Point (PDP):** This component serves as the centralised logic engine where policies are stored, interpreted, and evaluated. It receives decision requests from one or more PEPs, analyses these requests against the configured policy set (considering all relevant contextual data provided by the PEP, such as peer identity, proposed algorithms,

time of day, etc.), and returns a clear, actionable decision (e.g., Permit, Deny, Permit with specific cryptographic parameters) [47, 48]. Crucially, the PDP is decoupled from the enforcement mechanism. This decoupling allows policies to be managed and updated centrally without modifying the PEPs, providing significant operational flexibility and agility. While historically implemented using various technologies, modern PDPs often leverage dedicated policy engines.

Optionally, this core model can be augmented by other components often found in comprehensive policy frameworks like XACML [48]:

- **Policy Information Point (Policy Information Point (PIP)):** Responsible for retrieving additional attributes or context needed for policy evaluation (e.g., user roles from an LDAP directory, device posture information) that might not be directly available to the PEP.
- **Policy Administration Point (Policy Administration Point (PAP)):** The component or interface used by administrators to author, manage, store, and distribute policies to the PDP.

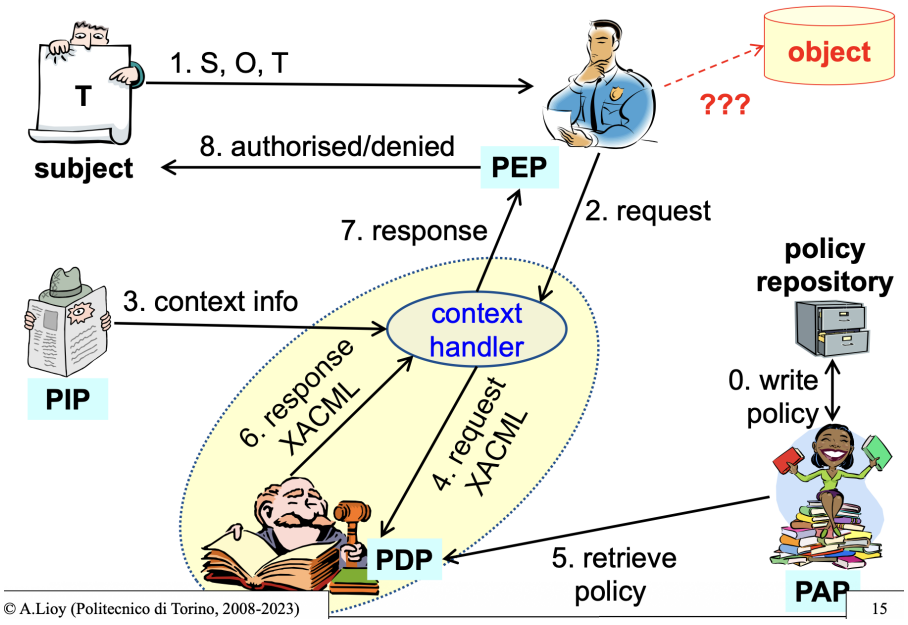

Figure 2.5. Reference architecture for policy-based access control, illustrating the roles of PEP, PDP, PIP, and PAP, inspired by the XACML model (Adapted from [48]).

While PIP and PAP are important for a complete system, the fundamental interaction enabling crypto-agility in our gateway focuses on the dynamic dialogue between strongSwan (PEP) and an external PDP. This architecture allows us to move the complex logic of selecting appropriate PQC algorithms, enforcing hybrid modes, or reacting to newly discovered vulnerabilities out of strongSwan's core configuration files and into a dedicated, more flexible policy engine.

Open Policy Agent (OPA) as a Modern PDP Implementation

To implement the PDP role effectively, particularly in a cloud-native or modern infrastructure context, a versatile and powerful policy engine is required. **Open Policy Agent (OPA)** [49] has gained significant prominence as an open-source, general-purpose policy engine designed precisely for this kind of decoupled policy decision-making. OPA is not specific to network security but can enforce policies across a vast array of software systems, including microservices, Kubernetes clusters, Continuous Integration/Continuous Deployment (CI/CD) pipelines, and, crucially for us, custom integrations with network gateways like strongSwan [50].

Several key features make OPA an excellent choice for implementing the PDP in our PQC-agile architecture:

- **Declarative Policy Language (Rego):** OPA policies are written in Rego, a high-level language designed specifically for expressing policies over complex, hierarchical data structures (like JavaScript Object Notation (JSON)). Rego focuses on querying input data and existing policy data to produce decisions, making policies relatively straightforward to read, write, and reason about. This allows administrators to define rules like, "For an IKEv2 connection, if the peer proposes `ML_KEM_768_IPD` and `ECDH_P384`, the decision is 'permit' and the selected algorithms must include both," in a clear, declarative manner, separate from the enforcement code.

- **Contextual Decision-Making with Structured Data:** OPA excels at consuming arbitrary JSON (or YAML Ain't Markup Language (YAML)) data as input to its policy evaluations. This is ideal for our use case, where strongSwan (the PEP) can formulate a detailed JSON object containing all relevant information from the IKEv2 negotiation (e.g., proposed transforms for encryption, integrity, PRF, classical Key Exchange Payload (KE), PQC KEM, peer identity certificates, source Internet Protocol (IP) address) and send it to OPA. OPA's Rego policies can then parse this rich context to make fine-grained decisions, far beyond simple allow/deny verdicts.

- **Decoupling and Agility:** OPA embodies the PDP/PEP separation. strongSwan only needs to know how to query OPA (typically via a simple RESTful Application Programming Interface (API) call) and how to interpret the JSON decision returned. All the complex logic regarding algorithm selection, hybrid mode enforcement, risk assessment based on peer identity, or cryptographic deprecation resides within OPA's Rego policies. These policies can be updated dynamically, often without restarting strongSwan, providing the essential mechanism for cryptographic agility. If a vulnerability is found in, say, `ML_KEM_512_IPD`, an administrator can update the OPA policy to disallow its negotiation, and this change takes effect immediately across all gateways querying that OPA instance.

- **Performance and Deployment Flexibility:** OPA is designed to be lightweight and performant. Policies can be compiled for efficient evaluation. It can be run as a standalone daemon, a sidecar container, or even embedded as a library, offering flexibility to suit different deployment architectures and latency requirements. For an IPsec gateway, running OPA as a co-located service often provides the necessary low-latency responses for real-time IKEv2 decisions.

By leveraging the standardised PDP/PEP architecture and employing OPA with its powerful Rego language as the PDP, we can construct a strongSwan-based gateway where the selection and enforcement of cryptographic algorithms, including complex PQC and hybrid configurations, are governed by external, dynamically updatable policies. This architecture forms the core of the crypto-agile solution developed in this thesis.

2.4 Performance of PQC in IKEv2: A Literature Review

The transition to PQC is not purely a question of security; it is also one of practical performance. The algorithms discussed in Section 2.1.3 present significant operational trade-offs compared to their classical ECC counterparts. Well understanding these trade-offs is essential, as they directly influence which algorithms can be deployed and serve as the baseline for the experimental validation in Chapter 4 of this thesis.

The academic literature has focused on quantifying this overhead, primarily using the same software stack as this project: strongSwan [41] integrated with `liboqs` [45]. The performance impact can be analysed in two main categories: computational overhead and bandwidth overhead.

Computational Overhead (Latency) For a latency-sensitive protocol like IKEv2 [42], the computational cost of the key exchange (`IKESA_INIT` [42]) and authentication (`IKE_AUTH` [42]) is critical. Multiple studies have benchmarked this, reaching a broad consensus [51, 52]:

- **Lattice-based KEMs (Kyber/ML-KEM):** Counter-intuitively, the primary PQC KEMs are exceptionally fast. Several analyses show that ML-KEM (Kyber) often outperforms high-security classical ECC (like P-521) and is at least comparable to the common P-256 [51]. This means that for the key exchange, the computational bottleneck is not PQC.
- **Lattice-based Signatures (Dilithium/ML-DSA):** Similarly, ML-DSA is computationally very efficient for both signing and verifying.
- **Hash-based Signatures (SPHINCS+/SLH-DSA):** This is the exception. While verification is fast, the *signing* operation is computationally intensive and slow [52]. This directly impacts the gateway (as the responder in the `IKE_AUTH` exchange), which must perform this slow operation, potentially increasing handshake latency or opening a vector for Denial of Service (DoS) attacks.

Bandwidth Overhead (Fragmentation) The most significant and problematic finding in the literature is not computation, but **data size** [53]. As introduced in Section 2.1.4, PQC public keys and signatures are orders of magnitude larger than their classical counterparts.

This directly impacts the IKEv2 protocol [42], which runs over UDP. The `KE`, `CERT`, and `AUTH` payloads become so large that the IKEv2 messages exceed the standard Ethernet Maximum Transmission Unit (MTU) (1500 bytes), resulting in **IP fragmentation** [51, 53]. This is a critical, practical failure point. Many firewalls, NAT devices, and network providers are configured to drop fragmented UDP packets as a security measure. Consequently, a PQC-enabled IKEv2 handshake may fail entirely, not due to a cryptographic error, but due to incompatible network infrastructure [53].

This confirms that that a successful PQC migration is not as simple as "swapping" algorithms. It requires an agile architecture, like the one proposed in this thesis, that can make intelligent, policy-based decisions. Such a system must be able to manage the trade-offs between the high-security but large SLH-DSA and the fast, smaller ML-DSA, and potentially negotiate different suites based on network paths or peer capabilities to avoid failures from fragmentation.

Chapter 3

Design of the PQC-Agile Gateway

Chapter 2 closed by translating the quantum threat into a concrete set of technologies and standards: lattice-based KEMs and signatures now formalised by NIST, the RFC extensions that carry them within IKEv2, the strongSwan architecture that handles modern IPsec, and the policy engines capable of externalising trust. The purpose of this chapter is to map that theoretical foundation onto the artefacts that constitute our system. We move from "what must be done" to "how the gateway is structured so that it can do it", focusing on architectural commitments, policy partitioning, and the control- and data-plane contracts that bind the implementation together.

3.1 From Requirements to Design Principles

The design phase began by aligning the abstract requirements captured in the previous chapters with the practical constraints of our deployment. Each principle documented below corresponds to tangible code or configuration that appears later in this thesis, ensuring that the logical model is traceable all the way down to the implementation described in Chapter 4.

3.1.1 Security Objectives

The first objective is to mitigate the "harvest now, decrypt later" threat by ensuring that every IKE negotiation is checked against a quantum-aware policy before an SA reaches the kernel. Operationally, the gateway prefers the PQC suites offered by the peer, and the policy engine can encode thresholds such as the minimum NIST security level or explicit whitelists and blacklists. Sessions that fail these rules are rejected during negotiation, which prevents the quiet regression that often accompanies transitional deployments [1].

The second objective is resilience. Confidentiality and authenticity must survive even if a single primitive is broken in future, as emphasised in Section 2.1.5. For peers that support it, the policy can insist on hybrid mode so that the derived keys inherit strength from both classical and lattice-based contributions. Rekeying follows the same approach, keeping the negotiated posture over time so that compromise of one family does not cascade into a systemic failure.

The third objective is accountability. Embedding trust choices in `swanctl.conf` would not give administrators the speed or audit trail required in the PQC transition. Instead, strongSwan acts as a PEP and delegates the final decision to Open Policy Agent as the PDP. The patched ext-auth listener extracts the runtime proposal, forwards it to OPA, and records the outcome together with the policy rationale. Every permit, deny, or forced renegotiation can therefore be traced back to an explicit declarative rule, which fulfils both compliance and operational needs.

3.1.2 Functional Scope

The gateway fulfils its mandate by handling two complementary families of flows that traverse exactly the same enforcement stack. Inbound traffic originates on the partner side, terminates

on the enforcement gateway, and is delivered to the internal services once policy approval is obtained. Outbound traffic follows the reverse path: it starts on the legacy estate, is normalised through the same controls, and reaches the partner environments only when the policy allows it. This symmetry ensures that a single declarative policy governs both directions, even when the underlying selectors and forwarding logic diverge. Chapter 4 will describe how these abstract responsibilities map onto concrete containers and network segments.

The control plane acts as the liaison between requirements and enforcement. The external authorisation hook captures every IKE negotiation, enriches it with certificate and context metadata, and submits the bundle to Open Policy Agent. The suite publisher records the negotiated primitives so that the decision store can expose them to downstream consumers, while the child manager translates the policy response into deterministic CHILD_SA operations. Because each actor shares a consistent view of the negotiation, the resulting audit trail spans the policy decision, the negotiated suite, and the installed state inside strongSwan.

Hybrid readiness remains within scope for every connection profile. The gateway is able to advertise the additional key exchanges defined in Section 2.2.3 so that peers capable of lattice-based encapsulation can contribute it without bespoke overlays. When a peer falls back to purely classical primitives, the same policy path still records the downgrade and can instruct the gateway to proceed or to reject the session altogether. This design keeps the migration to post-quantum cryptography incremental and reversible.

Finally, legacy interoperability and observability are treated as first-class outcomes. Classical-only partners continue to authenticate with their existing credentials yet still traverse the external policy checks. The decision store feeds both real-time dashboards and automated validation harnesses, so the evidence collected during testing mirrors the artefacts available in operations. This closes the loop between the requirements asserted here and the practical workflow that sustains them, ready for the architectural breakdown that follows in the rest of Chapter 3.

3.1.3 Operational Constraints

The design must respect the boundaries imposed by a containerised deployment. The gateway, its helpers, the partner simulators, and the policy backends are orchestrated through the same compose stack that will be reused in Chapter 4, so every component has to tolerate rebuilds and restarts without manual intervention. The bootstrap script binds the public interface, reinstalls the static routes, and primes the credential stores each time the container comes up; design choices therefore favour deterministic configuration over ad hoc tuning.

Automation further constrains timing and state management. The strongSwan commands invoked by the validation harness expose timeouts that have already been tuned to match the behaviour of hybrid IKE handshakes. Any new logic introduced in the control plane must complete within those windows or be able to signal failure in a way that the harness can capture. Similarly, helper processes such as the suite publisher and the child manager assume that negotiation artefacts appear in a predictable order; the design avoids out-of-band mutations that would invalidate that assumption.

The decision store is the coordination point for asynchronous workers and for observability. Besides the policy verdicts (`ike-`$< id >$`.json`), it now holds the negotiated-suite cache (`suite-cache/`), pending extraction requests (`suite-requests/`), CHILD snapshots captured by `child_suite_cache.py` (`child-suites/`), and outbound provisioning state (`tunnel-provisioning/`). Components that write to or read from those directories must obey the naming convention and hand-off rules already established, otherwise downstream consumers would miss state transitions or report false anomalies; the same conventions feed the dashboards and forensic workflows described later in this chapter.

Finally, the audit and monitoring requirements dictate how the system surfaces its internal state. Operational tooling depends on structured logs emitted by the external authorisation hook, the child manager, and the decision logger; the design therefore keeps those channels authoritative and refrains from duplicating their content elsewhere. Any extension to the gateway must integrate with the same logging scheme so that incident responders can follow a single, consistent trail from the initial negotiation to the installed security association.

3.2 Logical Architecture

This section assembles the logical view of the gateway by enumerating the components that implement the control and policy layers. It highlights how the building blocks interact so that later sections can drill into flows, trust boundaries, and enforcement mechanics.

Dashed boxes denote logical groupings.

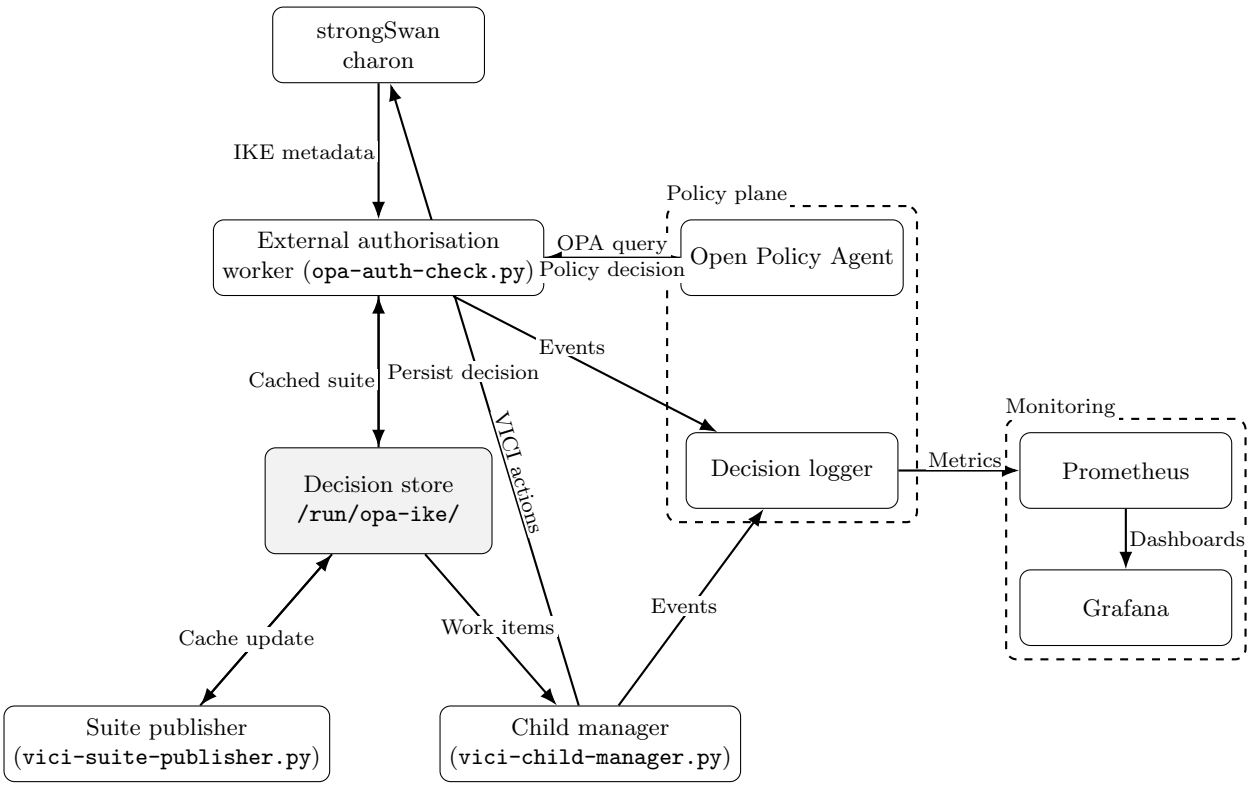

Figure 3.1. Logical architecture of the PQC-agile gateway, showing control, policy, and observability components.

3.2.1 Component Overview

The **strongSwan stack** remains the authoritative IKE and IPsec engine, with charon managing security associations and exposing lifecycle hooks that the rest of the system relies upon. The daemon operates with the standard plugin set and relies on the patched external authorisation interface to delegate policy decisions instead of embedding static rules inside `swanctl.conf`.

The **external authorisation worker** is implemented by the Python script `opa-auth-check.py`. It runs inside the gateway container, gathers the negotiation context via the `PLUTO_*` environment variables, and orchestrates suite extraction before calling Open Policy Agent. Its output is the decision file stored under the shared state directory that other helpers consume.

The patched listener also captures the "OPA_HINT" lines that the hook emits whenever a negotiation is denied. Each hint stores the IKE unique identifier together with the minimum key-exchange and certificate levels that OPA demanded, so the listener can deliver a deterministic response back to the peer on the next IKE_AUTH. This notifier does not alter the enforcement path denied negotiations still fail closed-but it prevents initiators from guessing which profile they must adopt when the gateway tightens its policy.

Two dedicated **VICI-based helpers** insulate charon from long-running tasks. The `vici-suite-publisher.py` service resolves negotiated primitives either through VICI queries or by parsing the live charon log and then publishes cache entries for the authorisation hook. The `vici-child-manager.py` service watches the decision store, reconciles pending actions with the live IKE inventory, and initiates or tears down CHILD_SAs with deterministic reqids and bounded back-off. It also tracks the active IKE pair for each service and mapped peer, probes `/v1/data/rekey/ike_sa/decision` with the old-versus-new suites, and emits `REKEY_ALLOW` or `REKEY_DENY` events based on the OPA verdict. When the policy approves the upgrade the helper terminates the predecessor IKE and promotes the newcomer; when the verdict rejects a downgrade it kills the new IKE, keeps the previous one pinned in `/run/opa-ike/ike-`$< id >$`.json`, and records the failure for audit.

The **child-suite cache helper** is implemented by `child_suite_cache.py`. It subscribes to up/down notifications for each `CHILD_SA`, snapshots the AEAD and DH identifiers together with both traffic selectors, and persists JSON artefacts under `/run/opa-ike/child-suites/` so that verifiers can recover the exact suite even after the tunnel is torn down.

The companion **tunnel provisioner**, `vici-tunnel-provisioner.py`, scans the legacy connections exposed by `swanctl --list-sas`, queries the OPA policy `tunnel_provisioning.rego` for the outbound `gw-wan-out-*` profile to initiate, and records outcomes under `/run/opa-ike /tunnel-provisioning/`.

The **decision store** under `/run/opa-ike/` collects every artefact exchanged between synchronous and asynchronous workers: the OPA verdicts (`ike-`$< id >$`.json`), the negotiated suite cache (`suite-cache/`), pending extraction requests (`suite-requests/`), the per-CHILD ESP snapshots maintained by `child_suite_cache.py`, and the tunnel-provisioning state emitted by the outbound watcher (`tunnel-provisioning/`). Its structure lets helpers restart without losing context while keeping the working set bounded.

The **Open Policy Agent service** executes the declarative policies that drive the gateway. It runs as a dedicated container that exposes the REST endpoint used by the authorisation worker and loads policy bundles that encode cryptographic thresholds, traffic selectors, and certificate requirements.

The **decision logger** collects structured events emitted by the hook and by the child manager. It preserves an immutable audit trail, exports Prometheus metrics, and feeds the dashboards that operators use to cross-check policy evaluations against the installed state of the gateway.

The **monitoring layer** complements the control plane with visibility tooling. Prometheus scrapes the decision logger and other exporters, while Grafana provides prebuilt dashboards that correlate policy responses, suite choices, and tunnel status with the compose services that implement the gateway. Together they ensure that operational staff can trace an IKE negotiation from the original request through to the final enforcement action without leaving the monitored surface.

3.2.2 Interaction View

A typical inbound negotiation starts the moment charon selects a proposal for a partner peer and raises the external authorisation hook. The hook inspects the `PLUTO_*` environment, translates ephemeral identifiers into the canonical connection names, and immediately asks the suite publisher whether the negotiated primitives have already been cached. If the cache is empty the hook emits a suite request, but it does not block: instead it begins assembling the payload for Open Policy Agent using the certificate metadata, the negotiated suite (either retrieved instantly or read back from the cache on the next iteration), the traffic selectors, and the connection role. The result is a JSON document that captures the entire context of the IKE negotiation and is sent via HTTP to OPA. A positive policy response contains both the allow/deny verdict and the child profile that must be enforced; the hook persists that decision to `/run/opa-ike/`, removes any stale files with the same unique identifier, and queues an event for the decision logger so the audit trail starts with the exact payload accepted by the gateway.

The decision store is the rendezvous point for every component that contributes to enforcement. As soon as the authorisation hook writes `ike-<id>.json`, the child manager notices a new entry and checks whether it has already been installed. If not, it reconciles the decision with the live IKE inventory through VICI. Installations that require a childless phase first see the manager waiting for the base IKE to settle, then issuing the CREATE_CHILD_SA using the reqid range reserved for that profile. The same watchdog is responsible for retries: should the CHILD creation fail because the gateway and peer raced to install the tunnel, the manager retries with exponential backoff and records the outcome by rotating the state file to `.done` or `.failed`. Each transition produces a structured log entry containing the unique identifier, the command issued, and the reason for success or failure; these `CHILD_MANAGER_EVENT` lines remain in the local logs and are made available to dashboards via file tailing. When the CHILD comes up the up/down hook reads the cached AEAD/DH pair and traffic selectors from `/run/opa-ike/child-suites/` before invoking `child.create`, so the audit trail always references the exact suite that charon installed.

Inbound data-plane readiness is therefore a collaborative effort. The hook records the provenance of the OPA verdict; the suite publisher supplies cryptographic context; the child manager enforces the decision and confirms the kernel state; the decision logger captures every step as a machine-readable audit stream. Operators can replay this sequence after the fact by correlating the decision file stored in `/run/opa-ike/`, the suite cache entry produced under `/run/opa-ike/suite-cache/`, and the structured log entries ingested by Prometheus. Should an inbound negotiation fail, those artefacts reveal whether the fault lies with policy evaluation, suite extraction, VICI orchestration, or the peer itself.

Outbound flows reuse the exact same contracts even though the gateway acts as initiator. Legacy traffic is detected automatically by `vici-tunnel-provisioner.py`, which scans `swanctl --list-sas` for the classic `gateway-legacy-*` profiles, asks the OPA policy `tunnel_provisioning.rego` which `gw-wan-out-*` profile to raise, and seeds the decision store before calling `swanctl --initiate`. Manual tooling such as `new-outbound-decision.sh` remains available for ad hoc tests, but the steady-state path is now driven by the provisioner. The authorisation hook still invokes OPA, but this time the role is "initiator" and the peer metadata is derived from the outbound connection template rather than a remote certificate. Because the decision file already exists, the child manager does not wait for a new entry; instead it watches the state transition from `.json` to `.done` once the outbound child has been created successfully. Any transient failures-missing suite, slow VICI response, or partner rejection-follow the same retry and logging logic as inbound negotiations, keeping the operational model uniform across directions.

When OPA denies a negotiation the external authorisation worker also emits an "OPA_HINT" describing the required key-exchange and certificate posture. The listener caches that hint alongside the IKE unique identifier and, if the peer retries, attaches a vendor notify (ID "0xA001") to the next outbound IKE_AUTH with payload `required_ke=<KE-Lx>;cert=<SIG-Ly>`. Peers that understand the extension can adjust their suite or certificate before reattempting, while legacy peers ignore the notify without affecting interoperability-the enforcement decision remains governed entirely by the policy verdict stored under "/run/opa-ike/".

Rekeys reuse the same instrumentation. When strongSwan raises a new IKE_SA while the previous tuple for that service still exists, the child manager treats the pair as a candidate make-before-break event. It collects the PRF, DH, and hybrid KEM identifiers from both SAs, posts them together with service and peer metadata to `data.rekey.ike_sa.decision`, and waits for the verdict. An `allow` response produces a `REKEY_ALLOW` log, the manager terminates the predecessor, and the decision file keeps only the new identifier. A `deny` response causes a `REKEY_DENY` line, the helper deletes the new IKE, and the old SA remains active until the peer retries with compliant primitives.

The monitoring stack consumes the resulting event stream. The decision logger exposes high-cardinality labels such as peer name, service, policy outcome, and suite source, which Prometheus scrapes at regular intervals. Grafana dashboards then correlate those metrics with the raw decision files and the strongSwan state to provide a live view of which connections are pending, which have completed, and which have failed. Because every helper reports its own progress, an operator can see, for any given unique identifier, when the policy decision was recorded, when the suite was

published, when the child manager attempted installation, and whether the result propagated to the kernel. This interaction view therefore closes the loop between the policy engine, the enforcement components, and the observability layer, ensuring that each negotiation can be traced and reasoned about end to end.

3.2.3 Trust and Threat Posture

The gateway deliberately separates responsibility into three trust zones. The enforcement surface lives inside the pep-gateway container where strongSwan, the external authorisation hook, the suite publisher, the child-suite cache helper, the tunnel provisioner, and the child manager share the decision store under `/run/opa-ike/` (including `suite-cache/`, `suite-requests/`, `child-suites/`, and `tunnel-provisioning/`). That directory is created with root ownership and is not bind-mounted outside the container, so only the helper processes can touch the state files. Everything beyond that boundary-Open Policy Agent, the decision logger, the CTI bundle service, Prometheus, and Grafana-resides on the isolated "opa-network" bridge defined in `docker-compose.yml`. The policy plane (OPA + decision logger) trusts the gateway only to forward well-formed requests; it never assumes that local configuration is correct and therefore revalidates every suite, certificate, and traffic selector before returning an allow. During rekeys the child manager also consults the `rekey_ike_sa.rego` decision to ensure the new IKE suite meets or exceeds the service minimum before promoting it. External peers sit outside the trust domain completely: they authenticate with certificates rooted in the PQ gateway PKI, and any mismatch in issuer, service, or selector is denied before the kernel sees the CHILD_SA.

Security-hardening in the compose stack reinforces these boundaries. The OPA, decision-logger, CTI updater, and Prometheus containers all run as non-root users with `no-new-privileges=true`, `cap_drop: [ALL]`, and read-only filesystems backed by tmpfs scratch space. Prometheus and Grafana export only their HTTP dashboards; the policy plane is reachable exclusively on the internal bridge, so a compromised peer cannot talk to OPA directly. Within the gateway container, the authorisation hook is fail-closed: if suite extraction times out or OPA does not answer, the negotiation is denied and logged. The child manager persists ".done"/".failed" markers to prevent replay of stale decisions, while the verifier script pulls the cached AEAD/DH and selectors from `/run/opa-ike/child-suites/`, revalidates them with `child.create`, and (when `ENFORCE_TEARDOWN=true`) tears down any CHILD that drifts from policy.

Threat intelligence and policy bundles are treated as untrusted input even though they originate from internal services. The CTI unified service signs each bundle with JWT ES512 and exposes it behind bearer-token authentication, and OPA validates those signatures before loading new data. At evaluation time OPA blends the CTI verdicts with the peer certificate and negotiated suite to enforce minimum KE/SIG levels per service. That approach mitigates harvest-now-decrypt-later risks by refusing to settle for weaker algorithms and rejecting rekeys that would downgrade the tunnel. Operational tooling closes the loop: the decision logger emits immutable events consumed by Prometheus, and Grafana dashboards correlate decisions, suite selections, and rekeys so that operators can spot anomalies (unexpected denials, repeated retries, suspicious source IPs) and respond before they turn into incidents. The tunnel provisioner and child-suite cache write their own structured logs under `/var/log/` so anomalies in outbound bootstrap or ESP capture appear alongside the existing ext-auth and child-manager streams.

Residual risk is explicitly bounded. If an attacker breached the gateway container, they could attempt to tamper with `/run/opa-ike/`, but the helper processes would record the inconsistency in their structured logs and Prometheus metrics (missing ".done" files, duplicate reqids, or verifier failures). If OPA were compromised, the fail-closed enforcement path would still prevent an unapproved tunnel from reaching the kernel, and the audit trail would show the unexpected decisions. Finally, policy updates and CTI bundles are versioned and logged, so every change to the trust policy is traceable and can be rolled back if a regression is detected.

3.3 Crypto-Agility Strategy

3.3.1 Algorithm Portfolio and Hybrid Policy Intent

The gateway negotiates hybrid suites that combine classical DH with ML-KEM while keeping BIKE and HQC in reserve for CHILD SAs. In `pep-gateway/swanctl/swanctl.conf` the proposals appear with the strongSwan naming (`ke1_kyber{1,3,5}` etc.), but the policy bundle expresses the same suites as `ML-KEM-{512,768,1024}` and relies on `crypto_mapper.py` to translate between the two. Child templates defined in `policy/cti/child_templates.rego` expand each level into full multi-KEM ESP combinations (ML-KEM + BIKE + HQC) so that the verifier can check every component. The suite publisher still snapshots the negotiated PRF/DH/`ADDKE` tuple into `/run/opa-ike/suite-cache/`, ensuring OPA receives the exact primitives that charon accepted.

Security levels remain defined by the policy (Table 3.1), but they now drive several consumers: `ike_establishment.rego` classifies the IKE suite for the ext-auth hook, `child_create.rego` maps the level to the canonical template, and `rekey_ike_sa.rego` ensures successor IKESA never regress. The child manager cross-references the level returned by OPA with the StrongSwan templates (for example `inbound-legacy-pay-L3` or `to-bankA-L3`) and refuses to install a profile that does not match the requirement; during rekey it also obeys the `action` returned by `rekey_ike_sa` so that KE-L3 tunnels cannot be replaced by KE-L2. As before, both achieved and required levels are written to the decision logger so operators can audit the cryptographic posture over time.

Table 3.1. Key exchange levels enforced by the gateway

Level	Classical core	PQC additions	Services
KE-L1	`SHA-256`; `ECP-256` or `X25519`	`ML-KEM-512`; optional `BIKE-1`/`HQC-128`	HR,low-risk tests
KE-L2	`SHA-384`; `ECP-256` or `ECP-384`	`ML-KEM-512/768`; `BIKE-1/3`; `HQC-128/192`	ERP partners
KE-L3	`SHA-384`; `ECP-384` or `ECP-521`	`ML-KEM-768`; `BIKE-3`; `HQC-192`	Payments (default)
KE-L4	`SHA-384`; `ECP-521`	`ML-KEM-1024`; `BIKE-5`; `HQC-256`	Payments (elevated); guest

Certificate validation follows the same layered approach. The policy classifies signature schemes into signature levels (SIG-L1 to SIG-L4-SUF) based on the ML-DSA variant and any classical companion algorithm. Payments insist on SIG-L3-SUF (`ML-DSA-87` with `Ed448`), ERP requires SIG-L2, and HR accepts SIG-L1. `opa-auth-check.py` extracts certificate metadata at runtime, normalises issuer names, and forwards the OID to OPA; the policy verifies both the signature algorithm and the subject DN against the service-specific allow-lists. If a peer presents a lower-grade certificate or an unexpected issuer, the decision logger records `sig_level_insufficient` and the hook denies the negotiation, ensuring that PKI regressions cannot slip through configuration drift.

Hybrid intent is therefore enforced end to end. The strongSwan templates expose the full hybrid portfolio so that capable peers can bring `ML-KEM`, `BIKE`, and `HQC` to the table without manual changes. The policy engine codifies the intent by translating business requirements (which partner accesses which service) into minimum KE and SIG levels, while also flagging attempted downgrades, revoked issuers, or suites that fall outside the approved combinations. Because the decision logger captures both the achieved and required levels for every negotiation, operators can audit uptake of PQC algorithms and confirm that the migration stays aligned with the defined roadmap. This structure keeps the gateway adaptable to future algorithm updates: adding a new KEM or raising a service requirement only requires updating the policy tables and the associated strongSwan proposals, leaving the rest of the control plane unchanged.

3.3.2 Policy Segmentation

The policy repository is split into orthogonal packages so that cryptographic requirements can evolve without touching identity checks or automation rules. `policy/cti/ike_establishment.rego` enforces service-level minima (KE and SIG), applies CTI verdicts, and derives peer context from the subnet metadata in `service_classes.rego` and `peer_mapping.rego`. The companion module `policy/cti/child_create.rego` consumes the canonical child templates defined in `child_templates.rego`, checking that the template chosen by the gateway matches the authorised level and that the traffic selectors correspond to the subnet described in the routing documentation. Additional packages handle certificate parsing (`certificate_validation.rego`), dynamic outbound provisioning (`tunnel_provisioning.rego`), and downgrade-safe rekeys (`rekey_ike_sa.rego`), each focused on its own concern.

Each package exposes a small interface consumed by the helpers. During the establishment phase the external authorisation worker calls `data.ike.establishment.decision` and receives `allow`, `child_profile`, and diagnostic fields such as `required_ke_level` and `sig_level_achieved`. The child manager and verifier invoke `data.child.create.decision` and `data.child.create.enforce` to replay the cached configuration before creating or deleting a CHILD_SA, while the `data.rekey.ike_sa.decision` endpoint returns `allow`, `reason`, `action`, and `required_level` whenever two IKE tuples overlap during make-before-break. The tunnel provisioner queries `data.ike.tunnel_provisioning.decision` to discover which `gw-wan-out-*` profile to initiate whenever legacy traffic is detected, and `data.certificate_validation.inspect` stays callable on its own so that the CTI updater can pre-flight issuers without touching the rest of the bundle.

Policy segmentation also underpins the migration strategy. The KE/SIG tables reference the change log for historical decisions, so raising payments to KE-L4 requires only an update to the level mapping and the addition of the corresponding strongSwan template (a template for outbound trials is already available). Identity and routing rules stay untouched because they live in a different module. Similarly, adding a new partner means updating the issuer allow-list and the service ACLs without touching the cryptographic tables, while the CTI package can blacklist an address range on its own. Prometheus metrics exported by the decision logger mirror this segmentation: counters are labelled with `phase=establishment` or `phase=child_create`, and dashboards show which layer rejected a negotiation, making it clear whether a failure is cryptographic, identity, or threat-intel driven.

3.3.3 Decision Table Examples

The decision logger captures every evaluation in a structured line so that operators can replay what OPA decided and why. Table 3.2 summarises three representative scenarios observed during the latest validation runs.

Each row exposes the relationship between runtime inputs and the policy artefacts stored in the repository. The ext-auth hook collects the negotiated suite, certificate metadata, and connection role, then OPA validates those attributes against the KE/SIG tables defined in the policy bundle. When the decision is `allow`, the hook persists `ike-`$< id >$`.json` and the child manager progresses it to `.done` once the CHILD is installed. When the decision is `deny`, the file remains with the failure reason and the validation scripts surface the cause so that operators can respond. The Prometheus dashboards mirror the same segmentation: counters are labelled by `result`, `reason`, and `service`, allowing teams to prove that PQ-capable partners consistently land on the expected hybrid profiles while legacy peers are contained until they upgrade.

Table 3.2. Representative policy decisions captured by the decision logger

Scenario	**Context collected by ext-auth**	**Policy outcome**	**Evidence / notes**
PQ-capable partner (bankA inbound)	mlkem768, bike3, hqc192; SHA-384 + ECP-521; certificate SIG-L3-SUF; service payments	allow; profile inbound-legacy-pay-L3; KE level satisfied (KE-L3)	Logger: DECISION ALLOW ... KE: KE-L3, CERT_SIG: SIG-L3-SUF; child manager installs SA #27 (swanctl -list-sas)
Legacy-only peer (opsc inbound)	Classical DH ECP-256; no PQ addKE; certificate SIG.L1 (legacy ML-DSA-44)	deny; reason = ike_requirements _fail_KE-L1; negotiation aborted	Logger: DENY - reason = ike_requirements_fail; ext-auth logs "Fail-closed: OPA deny"; peer receives AUTHENTICATION_FAILED
Legacy flow triggers tunnel provisioner	IKE child detected on `gateway-legacy-auth`; legacy IP `10.200.0.3` talking to `198.51.100.10`; provisioner queries `tunnel_provisioning.rego`	allow; reason = Tunnel provisioning authorised; outbound IKE `gw-wan-out-bankA` initiated	Logger: `phase = tunnel_provisioning, result = allow`; provisioner log records `LEGACY_TRAFFIC_DETECTED`; decision store shows `tunnel-provisioning/`
Outbound guest rehearsal (to-bankA-L4)	Requested KE-L4 / SIG-L4-SUF; negotiated suite KE-L3; certificate SIG-L3-SUF (gateway initiator)	deny; reason *rekey_regression*; existing tunnel kept active	Logger: DENY - reason=*rekey_regression*; child manager leaves state without .done; metric `ike_decisions_total{result="deny",reason="rekey_regression"}` increments

3.4 Control Plane Design

3.4.1 Ext-auth Integration Blueprint

The external authorisation hook is introduced through the patch stored in the file `multi-host-pep/pep-gateway/patches/ext-auth-runtime-cert.patch`. That patch augments the strongSwan ext-auth plugin so that every IKE negotiation triggers the Python entrypoint `opa-auth-check.py` with the full set of `PLUTO_*` variables. At runtime the script is invoked by charon just after the IKE_SA proposal is selected, giving us a synchronous point in which to apply policy before the kernel sees any CHILD_SA.

The same listener now relays policy feedback back to the peer. When `opa-auth-check.py` denies a negotiation it logs an `OPA_HINT` line containing the gateway's unique identifier, the minimum key-exchange level, and the certificate level required by OPA. The patched listener

parses that hint, caches both thresholds, and, if the negotiation remains in a denied state, attaches a vendor-specific notify (0xA001) to the next outbound IKE_AUTH with payload required_ke=<level>;cert=<SIG-Lx>. As a result, the initiator immediately learns which posture it must adopt before retrying, avoiding blind guesswork.

The bootstrap script multi-host-pep/pep-gateway/startup.sh ensures that the hook environment is prepared each time the container starts. It discovers the public interface, refreshes the PKI material under /usr/local/etc/swanctl/, and exports the variables that the hook expects (for example OPA_URL, OPA_SUITE_CACHE_DIR, and OPA_SUITE_REQUEST_DIR). The Dockerfile adds the script and its dependencies into the gateway image, pins the Python interpreter, and sets the entrypoint so that charon loads the patched plugin when the container comes up, then starts vici-child-manager.py, vici-suite-publisher.py, vici-tunnel-provisioner.py, and child_suite_cache.py so that outbound provisioning and CHILD-suite capture are already running before traffic arrives.

When charon calls the hook, opa-auth-check.py collects the negotiation context, normalises host names, and extracts certificate metadata using openssl. It produces a structured payload that contains the negotiated suite, the peer address, the service derived from the connection template, and the certificate OID. Before contacting OPA the script checks the suite cache under /run/opa-ike/suite-cache/; if the cache is empty it emits a suite request to be resolved by the background publisher.

The HTTP request to OPA is issued against the /v1/data/ike/establishment/decision endpoint. The response is persisted to /run/opa-ike/ike-<id>.json along with the policy rationale and the computed child profile. If anything fails, OPA timeout, missing suite, malformed payload, the script denies the negotiation, logs the reason to /var/log/ext-auth.log, and exits with a non-zero status so that charon aborts the IKE_SA. Successful decisions append a record to the decision logger so that the observability stack can ingest the outcome over HTTP.

This blueprint keeps the integration deterministic: the patch guarantees that every negotiation goes through the Python layer; the startup script makes the environment predictable; the hook itself enforces a fail-closed posture while delegating the final verdict to OPA; the listener surfaces policy hints back to the peer via the vendor notify; and the decision store in /run/opa-ike/ provides the rendezvous point for the asynchronous workers described in the following sections.

3.4.2 Runtime Data Extraction and Policy Request

The ext-auth worker relies on the environment that charon exports for each negotiation. When the hook starts it reads the PLUTO_* variables, normalises peer identifiers, and maps connection names to the services. The helper functions of opa-auth-check.py collapse hostnames to lowercase, derive the logical service (payments, ERP, HR), and reconcile responder-side labels with the gateway naming convention.

Suite extraction follows the tiered approach. Before contacting the policy engine the script checks the cache under /run/opa-ike/suite-cache/. If the cache is empty it emits a request file in /run/opa-ike/suite-requests/; vici-suite-publisher.py watches that directory, queries charon via VICI, and tails /var/log/charon.log as a fallback. Every attempt is logged with the data source (cache, VICI, or charon log), and the hook simply waits for the publisher to fulfil the request; vici-suite-publisher.py polls the entry until SUITE_PUBLISHER_REQUEST_TTL expires (25 s by default), sleeping 250 ms between checks and tailing /var/log/charon.log if VICI stalls. When the TTL elapses the helper removes the stale request, which causes the hook to fail closed and log a suite_timeout.

Once the suite is available the hook assembles the JSON payload for Open Policy Agent. The payload includes the IKE role, peer address, service, negotiated PRF/DH/ADDKE values, certificate metadata extracted via openssl, and traffic selectors parsed from PLUTO_*. The request is sent to /v1/data/ike/establishment/decision with the timeout configured through OPA_TIMEOUT. OPA responds with the verdict, the child profile, and diagnostic fields such as the achieved level, required_ke_level, and required_cert_level; the hook persists the full structure to /run/opa-ike/ike-<id>.json, leaves the suite cache entry in place for downstream

helpers, and posts a structured record to the decision logger. Before finalising an `allow`, the hook now converts the `child_sa_config` returned by OPA into the StrongSwan naming (via `crypto_mapper.py`) and calls `validate_child_config_match(..., strict=true)` to compare it against the actual entry in `swanctl.conf`. Any mismatch in ESP proposals, traffic selectors, or auxiliary fields causes the hook to override OPA's verdict with `child_config_mismatch`, ensuring the repository configuration and the policy bundle remain in lockstep.

If OPA denies the request or the HTTP call times out the script removes any pending suite request, writes the failure reason to `/var/log/ext-auth.log`, and exits with a non-zero status so that charon aborts the negotiation. Before returning, the hook extracts the minimum KE and certificate levels from the response or the policy reason (functions `extract_required_ke_level()` and `extract_required_cert_level()`), then logs an `OPA_HINT` record with the unique identifier, both thresholds, and the affected service so that the listener can notify the peer.

This data path keeps policy evaluation deterministic while exposing enough instrumentation for troubleshooting. The suite cache and request directories provide clear artefacts for the suite publisher and child manager, the decision store snapshots every OPA response together with its diagnostic fields, and the ext-auth log records both successes and failures alongside the context that was sent to the policy engine.

3.4.3 Policy Decision Cycle and Enforcement

Once an establishment decision is written under `/run/opa-ike/`, the asynchronous helpers turn the policy intent into concrete state inside strongSwan. `vici-child-manager.py` polls the decision directory (functions around lines 640-760) and builds an in-memory queue of entries that are marked `allow` but are not yet paired with a `.done` or `.failed` suffix. For each entry the manager queries charon via VICI to confirm that the parent IKE_SA is active, compares the requested child profile with the live configuration, and issues the appropriate `--initiate` or `--terminate` through `swanctl`. Retries follow a bounded exponential backoff: counters are stored alongside the decision file, and a decision turns into `.failed` only after the configured number of attempts have been exhausted.

After a CHILD_SA succeeds, the manager renames the decision to `.done`, records the SPI pair, and emits a structured log entry such as `CHILD_MANAGER_EVENT "child": "...", "event": "INIT_SUCCESS"` to `/var/log/vici-child-manager.log`. If strongSwan reports a duplicate, selector mismatch, or negotiation failure, the manager preserves the original file with a `.failed` suffix and logs the reason, which the decision logger picks up as a denial with `phase = child_create`. The hook also triggers the verifier script (see `scripts/updown_verifier.py`) so that the installed selectors match the routing policy defined for each service.

The decision logger (`decision-logger/server.js`) ingests the establishment verdicts posted by `opa-auth-check.py` and the post-installation checks reported by `updown_verifier.py`, while the child manager contributes `CHILD_MANAGER_EVENT` lines that remain in the local logs. Each record includes the service, peer address, KE and SIG levels achieved, and the reason for success or failure. Prometheus scrapes these records via the `/logs` endpoint, exposing counters such as `ike_decisions_total{phase="establishment",result="allow"}` and `ike_decisions_total{phase="child_create",result="deny",reason="profile_mismatch"}`. Grafana dashboards reuse the same labels to show whether denials originated from the policy engine, from suite extraction, or from enforcement inside the gateway.

Downstream components therefore see a consistent life cycle. Establishment decisions originate in `opa-auth-check.py`, are evaluated by OPA, persisted under `/run/opa-ike/`, enforced by `vici-child-manager.py`, and finally recorded by the decision logger along with their Prometheus labels. If any step fails, the corresponding artefacts remain in place for inspection (decision file with `.failed`, child manager log, ext-auth log), allowing operators to retrace the entire policy decision cycle.

3.4.4 Failure Modes and Resilience

The deployment treats every failure path as a fail-closed event while leaving sufficient artefacts for diagnosis. If suite extraction cannot satisfy a request before `SUITE_PUBLISHER_REQUEST_TTL` elapses (25 s by default), `vici-suite-publisher.py` removes the corresponding stale entry from `/run/opa-ike/suite-requests/`. Then, the hook logs a `suite_timeout` before denying the negotiation, so the peer still receives `AUTHENTICATION_FAILED`. OPA outages follow the same pattern: the HTTP client honours `OPA_TIMEOUT`, and any non-200 response or connection error produces a denial tagged `reason=opa_timeout` in the decision logger, allowing operators to distinguish policy rejections from infrastructure incidents.

Downstream helpers continue the resilience chain. `vici-suite-publisher.py` alternates between VICI queries and charon-log parsing, so a transient VICI stall does not block the suite cache. `vici-child-manager.py` retries failed CREATE_CHILD_SA operations with exponential backoff, caps the number of attempts to avoid livelock, and renames the decision file to `.failed` once all tries are exhausted. This behaviour keeps the original payload available for forensic analysis while preventing the queue from looping on a hopeless negotiation. The up-down verifier enforces selector and profile integrity, rejecting any deviation even if strongSwan installs a tunnel, which protects against misconfiguration and malicious attempts to widen access.

Observability closes the loop. The decision logger exposes counters and structured events so that Prometheus can alert on repeated opaque failures such as suite extraction timeouts, OPA timeouts, or profile mismatches. Grafana dashboards correlate these metrics with the raw decision files in `/run/opa-ike/`, the suite cache entries, and the child manager log stream. Because every failure leaves artefacts in predictable locations ext-auth log, decision file, suite cache, child manager state operators can trace the issue back to its root cause and re-run the negotiation once the underlying fault has been resolved.

3.5 Data Plane and SA Layout

3.5.1 Naming and Traffic Selector Strategy

The layout of the CHILD_SA catalogue mirrors the physical topology shown in Figure 3.2. Three external partners (bankA, partnerB, opsC) each terminate a VPN tunnel on the gateway's WAN address `203.0.113.2/29`, which is reached via the edge router at `203.0.113.3`. On the inside, the gateway exposes three legacy hosts that represent the service enclaves: `10.200.0.3/32` for payments, `10.200.0.4/32` for ERP, and `10.200.0.5/32` for HR.

Every CHILD profile now advertises the shared legacy subnet (`10.200.0.0/24`) on the local side so that policy decisions apply to the whole enclave, while the remote selector stays pinned to a single partner host (`198.51.100.10/32`, `.11/32`, `.12/32`). This lets OPA authorise flows at subnet granularity yet keeps the peer view deterministic.

IKE parent profiles follow the convention `gw-wan-in-`$<$ *peer* $>$ for responder-side (inbound) tunnels and `gw-wan-out-`$<$ *peer* $>$ for initiator-side (outbound) tunnels. They publish only the WAN IP of the gateway and reserve unique reqids: values in the 100 range are used for inbound CHILD_SAs, values in the 200 range are used for outbound CHILD_SAs. This makes the output of `swanctl --list-sas` unambiguous: the parent name identifies who initiated the negotiation and the reqid range reveals the direction of the associated child.

CHILD profiles inherit the same structure. Inbound templates are named `inbound-legacy-`$<$ *service* $>$`-L`$<$ n $>$, where the service tag (pay, erp, hr) identifies the legacy subnet and the suffix `L`$<$ n $>$ matches the minimum key-exchange level enforced by policy. Outbound templates use `to-`$<$ *peer* $>$`-L`$<$ n $>$, binding the gateway to a specific partner host. On inbound profiles the `unique` flag is left unset because the child manager governs overlap during rekey, whereas outbound templates keep `unique = keep` so two generations can coexist while the tunnel provisioner promotes a successor. The figure highlights that local selectors are /24 while remote selectors remain /32.

For example, `inbound-legacy-pay-L3` and `to-bankA-L3` both map the payments subnet `10.200.0.0/24` to bankA's host `198.51.100.10/32`; `inbound-legacy-erp-L2` and `to-partnerB-L2` do the same for the ERP subnet `10.200.0.0/24` and `198.51.100.11/32`; `inbound-legacy-hr-L1` and `to-opsC-L1` connect the HR subnet `10.200.0.0/24` to `198.51.100.12/32`.

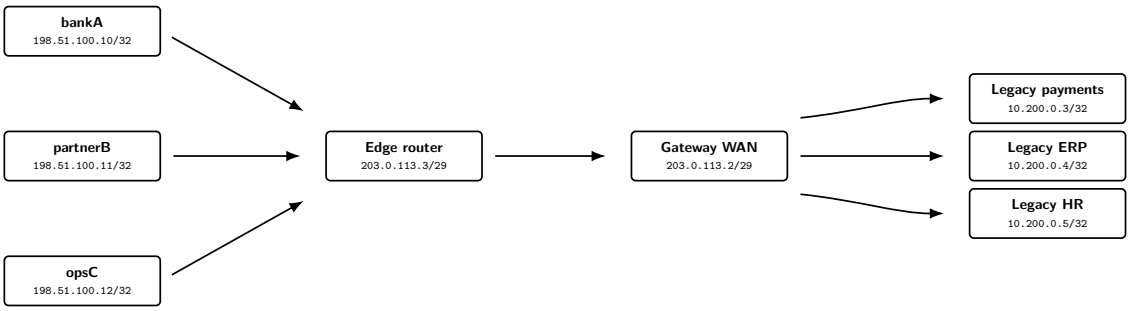

Figure 3.2. CHILD SA layout: each profile exposes the legacy subnet `10.200.0.0/24` to the partner host `198.51.100.x/32` through the gateway WAN address `203.0.113.2`.

3.5.2 Rekey and Lifetime Policy

The rekey cadence is embedded in the `pep-gateway/swanctl/swanctl.conf` templates. All inbound and outbound profiles inherit the same lifetime hierarchy: IKE SAs carry a shorter `rekeytime` than `lifetime`, giving the child manager time to install the successor before the kernel removes the predecessor. Outbound CHILD SAs retain `unique = keep` so two generations can coexist while traffic migrates, whereas inbound templates rely on the child manager (and the policy decision files) to coordinate make-before-break without setting that flag explicitly. Because the templates are shared across inbound and outbound scenarios, the gateway can renegotiate keys without forcing the peer to reconnect or disrupting state in the decision store.

Rekey events are handled cooperatively. When charon initiates a rekey, the external authorisation hook repeats the full OPA evaluation, writing the new verdict under `/run/opa-ike/ike-<id>.json`. `vici-child-manager.py` notices the new file, installs the successor CHILD, and leaves the previous decision marked as `.done` while the new file carries the updated state. If the rekey attempt weakens either the KE or SIG level compared with the existing tunnel, the policy returns `deny` and the manager preserves the original CHILD until the peer retries with acceptable parameters. This behaviour prevents downgrade attacks and ensures that a single slow partner does not block the rest of the site.

Failure handling mirrors the normal negotiation path. Should a rekey stall because the suite cannot be extracted, the hook removes the pending cache request and the child manager keeps the previous CHILD alive until the next attempt. Repeated failures show up in the decision logger as `reason=rekey_regression` or `reason=opa_timeout`, allowing operators to diagnose whether the issue stems from cryptographic policy or from a transient dependency. By keeping both SA generations in place during handover and by treating denials as non-destructive events, the gateway maintains continuous protection even when a peer renegotiates under load or over lossy links. Every make-before-break handshake feeds `data.rekey.ike_sa.decision` so that cryptographic downgrades or CTI flags prevent the successor from replacing the active tunnel. Upon approval the child manager logs `REKEY_ALLOW`, deletes the predecessor IKE, and keeps the new reqids, whereas a rejection emits `REKEY_DENY`, tears down the new SA, and leaves the earlier generation serving traffic until a compliant rekey arrives.

3.5.3 Observability and Audit Trail

Every negotiated SA leaves a paper trail across three layers: the control-plane logs, the decision store, and the metrics pipeline. During establishment the ext-auth hook writes a JSON decision file under `/run/opa-ike/` containing the peer address, service, negotiated

suite, policy verdict, child profile, and the thresholds returned by OPA (`required_ke_level` and `required_cert_level`). When the child manager installs (or rejects) the corresponding CHILD_SA it renames that file to `.done` or `.failed` and records a structured line in `/var/log/vici-child-manager.log`. These files form the primary audit artefacts: they can be replayed to understand which tunnel was requested, which policy rule was applied, and how the enforcement worker reacted.

The decision logger aggregates the establishment records posted by `opa-auth-check.py` and the post-installation checks emitted by `updown_verifier.py`. It also records `phase = rekey` decisions from `rekey_ike_sa.rego` and `phase = tunnel_provisioning` events whenever `vici-tunnel-provisioner.py` brings up an outbound profile.

Each OPA evaluation produces a record with `phase`, `result`, `reason`, `service`, `ke_level`, `sig_level`, `required_ke_level`, `required_cert_level`, and a timestamp; when the CHILD is raised or torn down, `updown_verifier.py` appends `phase = child_up` / `phase = child_down` entries with the final outcome, while the child manager keeps its `CHILD_MANAGER_EVENT` lines in the local logs. Prometheus scrapes the decision logger and exposes counters (e.g. `ike_decisions_total`), gauges (number of pending decisions), and latency histograms. Grafana dashboards combine those metrics with selective log excerpts so that operators can drill down from a spike in `reason=rekey_regression` to the exact JSON decision and, if needed, to the underlying suite-cache request that preceded it.

Operational tooling closes the loop on the data plane as well. Structured logs from `vici-tunnel-provisioner.py` and `child_suite_cache.py` are shipped in the same dashboard so operators can correlate outbound bootstrap attempts and ESP snapshots with the primary OPA decisions.

StrongSwan's `swanctl --list-sas` output is captured after automated test runs and stored alongside the decision files to prove that the installed CHILD selectors match the policy. The up/down verifier emits dedicated log lines whenever a mismatch is detected, ensuring that a tunnel cannot silently widen its selectors. Taken together, these artefacts provide a complete audit trail from policy decision to kernel-state installation and deliver enough signals for alerting, troubleshooting, and long-term compliance reporting.

3.6 Design Summary and Traceability

The design choices introduced throughout Chapter 3 map directly to the initial requirements: each objective is enforced by a specific architectural commitment, captured by tangible artefacts in the repository, and observable through the audit trail described earlier. Table 3.3 consolidates these links so reviewers can verify how the gateway satisfies every requirement and where the supporting evidence resides.

This synthesis shows how the OPA-driven control plane, the naming and selector discipline, the rekey policy, and the observability stack jointly deliver quantum-aware enforcement, resilience, and accountability while staying fully traceable to code and configuration.

Table 3.3. Traceability between requirements, design decisions, and implementation artefacts

Requirement	Design commitment	Implementation artefacts	Sections
Quantum-aware enforcement	Ext-auth hook defers every IKE_SA to OPA; hybrid profiles advertise ML-KEM/BIKE/HQC and enforce KE-L$\langle n \rangle$ levels	`scripts/opa-auth-check.py`, `pep-gateway/swanctl/swanctl.conf`, `policy/cti/ike_establishment.rego`, `policy/cti/certificate_validation.rego`	3.1.1, 3.3.1
Resilient negotiation and rekeying	Fail-closed error paths, suite cache with retries, child manager backoff, dual-generation rekeys (`unique = keep`)	`scripts/vici-suite-publisher.py`, `scripts/vici-child-manager.py`, `scripts/child_suite_cache.py`, decision files in `/run/opa-ike/`, lifetimes in `swanctl.conf`	3.4.4, 3.4.2, 3.5.2
Operator observability	Structured decision store, decision logger metrics (including `phase=tunnel_provisioning`), Prometheus/Grafana dashboards, up/down verifier for selectors	`decision-logger/server.js`, `/var/log/ext-auth.log`, `/var/log/vici-child-manager.log`, `/var/log/vici-tunnel-provisioner.log`, `/var/log/opa-ext-auth/updown-verifier.log`, `scripts/updown_verifier.py`	3.2.2, 3.5.3
Deterministic data-plane layout	Naming convention `gw-wan-in/out-*`, child profiles `inbound-legacy-*/to-*`, three fixed legacy subnets (`10.200.0.0/24` locals, state in `/run/opa-ike/tunnel-provisioning/`), partner selectors at /32	`pep-gateway/swanctl/swanctl.conf`, static routes in `pep-gateway/startup.sh`, Figure 3.2	3.5.1
Policy/code alignment	Repository patches embed ext-auth blueprint; Dockerfile/startup scripts preload environment variables and bind the decision store	`pep-gateway/patches/ext-auth-runtime-cert.patch`, `pep-gateway/Dockerfile`, `pep-gateway/startup.sh`, `docker-compose.yml`	3.4.1, 3.4.3

Chapter 4

Implementation

4.1 Implementation Overview

4.1.1 Compose Stack and Runtime Topology

The implementation lives inside a single `docker-compose.yml`, so the first step was to give the entire gateway its own self-contained runtime and to wire the supporting services around it. The compose file declares four bridge networks: `lan-net`, `wan-net`, `external-net`, and `opa-network`. They mirror the three zones introduced in Chapter 3 plus the partner segment. Each network carries a deterministic IP plan, which keeps the container addresses stable from one run to the next. That choice is deliberate; it means the configuration of strongSwan and the routing rules inside the containers never need to guess which interface or address they will get.

Service-wise, the stack is grouped into three clusters.

- **Enforcement plane:** `pep-gateway` is the strongSwan router, bound to `lan-net` and `wan-net` and, via the edge router, to the partner segment on `external-net`. The partner simulators (`banka`, `partnerb`, `opsc`) attach only to `external-net`, while the dedicated `edge-router` container bridges `wan-net` and `external-net` so that traffic follows exactly the L3 path documented earlier. The `host-legacy` container represents the internal initiator on `lan-net`.

- **Policy and monitoring plane:** Open Policy Agent (`opa`), the decision logger, the CTI bundle service, Prometheus, and Grafana all reside on `opa-network`. Keeping them off the enforcement bridges prevents an external peer from ever talking directly to the policy components and keeps the monitoring traffic isolated.

- **Shared state and telemetry:** consistent logging and bundle data are handled through dedicated volumes. The shared `logs` bind mount lets every component append to `/var/log` so that the validation scripts later in the thesis can collect artefacts from the host. Policy bundles, OPA data, and CTI cache use separate named volumes, which avoids cross-run contamination.

The compose file encodes the orchestration order through `depends_on`: policy services start first, the gateway waits for them, and only then do the partners come up. The result is a deterministic lab: bringing the stack online with `docker compose up -d` produces a gateway that already knows which networks it inhabits, which certificates it must load, and which helper scripts must run in the background. The next subsections describe how each of those components is built and how they interact.

4.1.2 Control-plane automation

The gateway relies on a thin set of Python helpers to keep the control plane predictable while charon negotiates IKE. Each helper plays a specific role and all of them follow the instrumentation conventions introduced in Chapter 3.

`opa-auth-check.py`: synchronous policy gate

`opa-auth-check.py` is invoked by the patched ext-auth plugin as soon as charon selects an IKE proposal. The script reads the full `PLUTO_*` context, normalises peer identifiers, and maps the connection name to the logical service described in the design chapter. It refreshes the strongSwan certificate chain on demand by shelling out to `openssl` so that the OPA payload always carries the current signature OID and validity window. Suite discovery follows a stepped path: the script first inspects `/run/opa-ike/suite-cache/`, then emits a request file under `/run/opa-ike/suite-requests/`, and finally waits for the publisher to fill the gap. Once the negotiated parameters are available the script assembles the JSON body for `/v1/data/ike/establishment/decision`, including role, service, negotiated PRF and hybrid KEs, certificate attributes, and the planned traffic selectors. A successful response is written to `/run/opa-ike/ike-`$< id >$`.json`, posted to the decision logger, and echoed in `/var/log/ext-auth.log` with a short status line. If OPA times out or denies the request the script exits with a non-zero status, ensuring that the gateway fails closed and leaves a reasoned artefact behind. Before returning it parses any `required_ke_level` or certificate hints so the notifier can feed the guidance back to the peer.

`vici-suite-publisher.py`: asynchronous suite resolver

`vici-suite-publisher.py` watches the suite-request directory and resolves every entry that the hook could not satisfy synchronously. It opens a VICI session against charon, queries the live IKE_SA by unique identifier, and persists the negotiated algorithms to `/run/opa-ike/suite-cache/ike-`$< id >$`.json`. When VICI stalls the worker tails `/var/log/charon.log` as a fallback, parsing the `selected proposal` lines and caching the tuple of PRF, DH, and post-quantum KEs. Each attempt is timestamped and logged under `/var/log/vici-suite-publisher.log` together with the source (cache, VICI, charon), the retry counter, and the elapsed time. The worker keeps polling the request file until the `SUITE_PUBLISHER_REQUEST_TTL` (25 s by default) expires, sleeping 250 ms between checks so latency and log volume stay bounded. By isolating suite extraction in this helper we avoid blocking the IKE thread, reduce contention on the VICI socket, and maintain a permanent record of the suites that were actually accepted in production.

`vici-child-manager.py`: decision enforcement

`vici-child-manager.py` polls `/run/opa-ike/` for files marked `"result": "allow"` that still lack a `.done` or `.failed` suffix. For each decision it checks that the parent IKE_SA is alive, reconciles the requested child profile with the static templates defined in `swanctl.conf`, and issues the matching `swanctl --initiate`. Inbound profiles use reqids in the 101–103 range, outbound ones start at 201, so the manager can detect duplicates and stale entries by simply reading the kernel view of the SAs. Retries follow a bounded exponential backoff and every attempt results in a structured `CHILD_MANAGER_EVENT` line with outcome, reqid, and SPI pair. On success the file gains a `.done` suffix and carries the issued SPIs for later audits. On failure the manager writes a `.failed` twin that preserves the policy body together with the final error, so operators can replay the same payload after fixing the underlying issue. The worker also looks after deletions: when OPA or the verifier requests a teardown the manager calls `swanctl --terminate` and records the result in the same log stream.

Rekeys follow the same code path. Whenever a new IKE_SA appears while the previous tuple for the same service and mapped peer is still alive, the helper collects both suites, posts the payload to `/v1/data/rekey/ike_sa/decision`, and waits for the `policy/cti/rekey_ike_sa.rego`

verdict. An allow response produces a `REKEY_ALLOW` event, the manager terminates the predecessor via VICI, and the decision file keeps only the successor's identifier, whereas a denial triggers `REKEY_DENY`, the new IKE_SA is torn down, and the old generation keeps servicing traffic until the peer retries with compliant primitives. The same logic preserves the `.json`/`.done`/`.failed` trail so auditors can replay which tuple was accepted and why.

`updown-verifier.py`: selector integrity checks

StrongSwan invokes `updown-verifier.sh`, a tiny wrapper that execs `updown_verifier.py`; the Python helper reads the up/down events and validates every installed CHILD against the selectors agreed during design. It reads the negotiated selectors and compares them against the authorised values stored in the decision file (`/run/opa-ike/ike-<id>.json`) and the cached ESP snapshot in `/run/opa-ike/child-suites/`; any widening or mismatch triggers an immediate teardown. The verifier records both successful inspections and mismatches under `/var/log/opa-ext-auth/updown-verifier.log`, tagging each line with service, peer, and reqid. These entries are shipped to the decision logger so that Prometheus and Grafana mirror the same pass or fail posture seen on the gateway. Although the script is short it closes the loop between design intent and live kernel state, which is why it sits alongside the heavier helpers in this subsection.

`child_suite_cache.py`: CHILD suite snapshots

`child_suite_cache.py` listens to the `child-updown` event stream, captures the AEAD/DH identifiers and both traffic selectors for every CHILD, and writes JSON artefacts under `/run/opa-ike/child-suites/`. By caching the suite independently of `swanctl --list-sas`, the verifier can retrieve the exact ESP parameters even after the tunnel is torn down, and the decision logger can link policy verdicts to the ciphers actually installed in the kernel.

`vici-tunnel-provisioner.py`: legacy-to-PQ bridge

`vici-tunnel-provisioner.py` scans the classic 'gateway-legacy-*' connections via `swanctl --list-sas`, detects when a legacy CHILD is active toward a partner, and asks the `tunnel_provisioning.rego` policy which outbound `gw-wan-out-*` profile to raise. When authorised it writes a marker under `/run/opa-ike/tunnel-provisioning/` and issues `swanctl --initiate`, ensuring PQ tunnels are created just-in-time without manual intervention; denial reasons (unapproved partner, insufficient level) are logged alongside the request.

4.1.3 Bootstrap order and dependencies

The `docker-compose.yml` encodes a strict startup sequence so the lab always boots into a known-good state. Policy services come first: Open Policy Agent, the decision logger, and the CTI bundle distributor are marked as prerequisites for the gateway, and each ships a health check that asserts the API is reachable before the rest of the stack proceeds. Only when those checks report "healthy" does Docker schedule `pep-gateway`, whose entrypoint `startup.sh` performs the routing bootstrap, loads the certificates, and starts the StrongSwan helpers. The edge router and the partner simulators ('banka', 'partnerb', and 'opsc') are chained after the gateway so their first IKE attempts always land on a ready responder instead of racing against the policy plane. Prometheus, Grafana, and the auxiliary exporters sit at the tail of the dependency graph; they start once the log directories exist and the decision logger exposes its '/logs' endpoint. This choreography prevents spurious failures during compose up: no peer timeouts while OPA loads its bundles, no missing decision store when the child manager comes alive, and no monitoring alerts caused by empty endpoints. A single `docker compose up -d` therefore yields a reproducible environment where every container knows its role, its peers, and the shared state it must mount before the validation scripts begin.

4.2 Gateway Build and Bootstrap

The gateway runs inside a dedicated container so every compose cycle reuses the same StrongSwan build, helper scripts, and credentials. The image is produced from `multi-host-pep/pep-gateway/Dockerfile`, while the runtime is initialised by `startup.sh` in the same directory.

4.2.1 Dockerfile pipeline

The Dockerfile is multi-stage and keeps both build and runtime layers on `ubuntu:22.04`. Stage one installs the toolchain (`build-essential`, `cmake`, `ninja-build`, OpenSSL headers, autotools) and compiles liboqs `0.10.0` with shared libraries so Dilithium, ML-KEM, BIKE, and HQC are available to StrongSwan. When the optional flag `ENABLE_OQS_PROVIDER=true` is set the same stage also builds oqs-provider `0.6.1` and places `oqsprovider.so` under `/usr/local/lib/ossl-modules/`. Stage two clones StrongSwan `6.0.0beta6`, applies `ext-auth-runtime-cert.patch`, runs `autogen.sh`, and configures the daemon with `--disable-ikev1`, `--enable-oqs`, `--enable-openssl`, `--enable-frodo`, `--enable-ext-auth`, and `--enable-vici` before installing it to `/usr/local/` and saving the configure log for audits. Stage three assembles the runtime image: it copies the StrongSwan tree and liboqs artefacts into `/usr/local/`, installs the networking utilities (iproute2, iptables, iputils, traceroute, tcpdump), and uses `pip` to add the Python dependencies `requests` and `vici`. The curated configuration set from `pep-gateway/swanctl/`, the patched `strongswan.conf`, and the helper scripts (`opa-auth-check.py`, `vici-child-manager.py`, `vici-suite-publisher.py`, `updown-verifier.sh`, `new-outbound-decision.sh`) are copied into place and marked executable. Environment defaults (`OPENSSL_MODULES`, `OPENSSL_CONF`, `LD_LIBRARY_PATH`, `OPA_URL`, `OPA_TIMEOUT`, `OPA_FAIL_OPEN`) are declared so operators can override them via compose without rebuilding the image. The image exposes UDP 500, UDP 4500, and TCP 4502, creates `/run/opa-ike` with restrictive permissions, and sets `/opt/gateway/bin/startup.sh` as entrypoint so every container boot performs the same preparation sequence before charon starts.

4.2.2 `startup.sh` responsibilities

`startup.sh` gives the container a deterministic boot path before StrongSwan begins handling IKE traffic. It first prints the policy endpoint in use, clears any stale `/var/run/charon.pid`, and exports `EXT_AUTH_DISABLE_VICI=0` so the patched ext-auth plugin can call back into VICI. The script launches `/usr/local/sbin/charon --use-syslog` in the background and records the PID to detect early failures. It then derives the public interface either from `GATEWAY_PUBLIC_IFACE` or by calling `discover_interface_for_ip` on `GATEWAY_PUBLIC_IP` and binds the /32 address with `ip addr add`. Once the WAN endpoint is in place it installs a static route towards `EDGE_EXTERNAL_SUBNET` via `EDGE_ROUTER_IP`, which matches the compose topology and keeps partner traffic flowing through the edge router. After a short grace period the script confirms that the charon PID is alive, warns if UDP 500 or UDP 4500 are not listening yet, and pushes the full configuration into the daemon with `swanctl --load-all`. The two Python helpers `vici-child-manager.py` and `vici-suite-publisher.py` are started via `/usr/bin/env python3`, inheriting the same environment so they can reach `/run/opa-ike`. Finally the script prints "IPsec Gateway ready with OPA automation" and tails `/dev/null` to keep the container running, leaving charon and the helpers to service negotiations. Every step mirrors the expectations encoded in the compose environment variables and guarantees that a fresh container reaches operational state before any test drives the control plane.

4.3 Data-plane Configuration

The runtime topology hinges on a set of strongSwan templates that keep every tunnel aligned with the routing plan introduced in Chapter 3. All responder and initiator profiles live under `multi-host-pep/pep-gateway/swanctl/swanctl.conf`, while each partner simulator loads

a mirrored file from `multi-host-pep/external-peers/peer/swanctl/swanctl.conf`. This section walks through the gateway catalogue, the partner-side configuration, and the glue that keeps selectors and routes symmetric across the WAN.

4.3.1 Gateway templates and naming discipline

Responder profiles follow the naming pattern `gw-wan-in-peer`, pinning `local_addrs = 203.0.113.2` and the matching `remote_addrs` drawn from the partner block (`198.51.100.10` for BankA, `198.51.100.11` for PartnerB, `198.51.100.12` for OpsC). Every inbound connection sets `childless = force` so the first IKE_SA comes up without payload selectors, leaving the child manager to install only the children sanctioned by OPA. Child definitions adopt the schema `inbound-legacy-service-Llevel`, advertise the legacy subnet `10.200.0.0/24` as the local selector, keep the partner host on /32, and attach deterministic reqids (101 through 105) that mark inbound flows. Outbound templates mirror the scheme with `gw-wan-out-peer` parents and child profiles titled `to-peer-Llevel`; they reuse the same /24 local selector, keep the partner on /32, and use reqids starting at 201 so tooling can tell direction at a glance. Only outbound templates set `unique = keep`; inbound ones leave the flag unset because the child manager (and the policy decision files) already govern make-before-break during rekeys. Certificate references point to the Dilithium chain staged under `/usr/local/etc/swanctl/x509/`, while classical counterparts remain available so that legacy scenarios (for example `gateway-legacy-auth`) can still be exercised without replumbing the repository. By centralising this naming discipline, the verifier, the up/down hook, and the decision store all speak the same language, making troubleshooting a matter of matching strings rather than translating between bespoke labels.

4.3.2 Partner endpoint configuration

Each partner simulator ships its own copy of `swanctl.conf` under `multi-host-pep/external-peers /`< *peer* >`/swanctl/`, fixing `local_addrs` to its host IP (`198.51.100.10`, `.11`, `.12`) and `remote_addrs` to the gateway's WAN `203.0.113.2`. The child profiles mirror the gateway naming (e.g. `peer-bankA-to-gw-wan`), reuse the same /24 subnet on the gateway side, and keep the partner selector at /32 so that selectors remain symmetric when the tunnel provisioner raises outbound flows. Each container mounts the shared Dilithium credentials read-only and sets `EXT_AUTH_DISABLE=1` so the patched hook stays focused on the gateway; static routes ('ip route replace 203.0.113.0/29 via 198.51.100.2' and 'ip route replace 10.200.0.0/16 via 198.51.100.2') are programmed at boot so all partner traffic transits the edge router at `198.51.100.2`. Centralising the partner configs here keeps the test harness deterministic: the same template can be exercised as responder or initiator simply by pointing `swanctl --initiate` at the relevant profile.

4.3.3 Selector symmetry and routing glue

Selector symmetry is enforced end to end: every inbound template publishes the legacy subnet `10.200.0.0/24` as the local TS while every partner template pins its remote selector to `198.51.100.x/32`, so the verifier and OPA see identical tuples regardless of direction. Static routes keep the overlay deterministic: `startup.sh` programs `ip route add 198.51.100.0/28 via 203.0.113.3` on the gateway, while each partner container replaces routes toward `203.0.113.0/29` and `10.200.0.0/16` via the edge router at `198.51.100.2`, ensuring all traffic crosses the same L3 hop documented in Chapter 3. Because both sides honour the same naming discipline and routing glue, the child manager, tunnel provisioner, and verifier can reason about requests purely from the profile name and the cached selectors, reducing the risk of asymmetric configurations.

4.3.4 Policy and decision-store integration

All helpers share `/run/opa-ike/` as the rendezvous point. `startup.sh` recreates the directory tree on every boot with root ownership, ensuring the state remains ephemeral yet ready for the Python

workers. `opa-auth-check.py` writes `ike-`$< id >$`.json` verdicts, `vici-child-manager.py` rotates them to `.done` or `.failed`, `vici-suite-publisher.py` populates `suite-cache/` and drains `suite-requests/`, `child_suite_cache.py` appends ESP artefacts under `child-suites/`, and `vici-tunnel-provisioner.py` leaves breadcrumbs inside `tunnel-provisioning/`. Make-before-break tracking reuses the same namespace: every overlapping IKE pair is logged side-by-side until the rekey policy decides whether to promote the newcomer or keep the predecessor. Because each directory has a single writer and a predictable naming scheme, log shippers and validation scripts can scrape the files straight from the host bind mount without racing the helpers.

4.4 Policy, CTI, and Decision Store

4.4.1 Rego modules and service mapping

The policy bundle under `policy/cti/` is organised so that every control-plane phase consults only the logic it needs while still sharing the same data sources. `ike_establishment.rego` receives the payload assembled by `opa-auth-check.py`, blends the certificate metadata with the peer catalogue maintained in `peer_mapping.rego`, looks up the target service in `service_classes.rego`, and evaluates the negotiated PRF/DH/PQC tuple against the KE/SIG tables; it returns `allow`, the canonical child profile, and diagnostic fields (`required_ke_level`, `sig_level_achieved`, CTI flags) so that the hook and decision logger can preserve the rationale. `child_create.rego` is invoked asynchronously by `vici-child-manager.py` before every CREATE_CHILD_SA and again by `updown_verifier.py`: it replays the selectors and template details recorded in `swanctl.conf`, ensuring that stale or tampered decision files cannot install a CHILD whose CIDRs or ESP proposals diverge from the approved profile. `rekey_ike_sa.rego` compares the suites of overlapping IKE generations during make-before-break, applies the same KE tables plus downgrade matrices for ML-KEM, BIKE, and HQC, and emits an `action` field (`promote_new_delete_old` or `terminate_new_keep_old`) that the child manager logs as `REKEY_ALLOW/REKEY_DENY`. `certificate_validation.rego` isolates the parsing of issuer DNs and signature OIDs harvested at runtime, grouping them by trust level so that new issuers or revoked signers can be rolled out independently of the rest of the policy. Finally, `tunnel_provisioning.rego` and its helpers ingest the outbound telemetry that `vici-tunnel-provisioner.py` writes under `/run/opa-ike/tunnel-provisioning/` and authorise only the `gw-wan-out-*` profile that matches the detected legacy flow, which keeps automated initiations aligned with the same ACLs enforced inbound. Together these modules let the compose lab raise or deny tunnels based on declarative requirements while keeping the interfaces between helpers small and well-defined.

4.4.2 CTI bundle distribution and verification

Threat intelligence arrives through the dedicated `cti-unified-service` container, which polls upstream feeds, normalises the indicators, and publishes signed bundles under `/bundles/cti/`. Each bundle carries semantic metadata (version, generation timestamp, source hashes) plus the consolidated artefacts themselves: malicious IPv4/v6 ranges, TOR exit nodes, revoked certificate issuers, partner-specific overrides, and manual quarantines that operators can raise during an incident. Before exposing a bundle the service composes the JSON tree, signs it with JWT ES512, and serves it over HTTPS behind bearer-token authentication; the compose file keeps this service ahead of the gateway so that fresh intelligence is available before any IKE negotiation starts. OPA mounts the bundle directory read-only, fetches the manifest at start-up, and refuses to load a revision unless the signature matches the trusted public key baked into the image, which prevents a compromised updater from injecting arbitrary data. Once validated, the indicators become part of the `data.cti` namespace that `ike_establishment.rego` and `rekey_ike_sa.rego` consult: malicious peers trigger `reason = threat_ip_blocked`, TOR exits downgrade guest access, and revoked issuers cause `sig_level_insufficient` even if the certificate chain would otherwise pass. Health checks on the CTI container expose the bundle age via Prometheus so dashboards can alert whenever the feed drifts beyond the acceptable staleness window, keeping the lab faithful to the operational posture it emulates.

4.4.3 Decision store layout

The shared directory under `/run/opa-ike/` acts as the rendezvous point for every helper and mirrors the life cycle of an IKE negotiation. Fresh OPA verdicts arrive as `ike-`$< id >$`.json` files containing the peer metadata, negotiated suite, diagnostic fields, and the child profile that must be enforced; once `vici-child-manager.py` succeeds it renames them to `ike-`$< id >$`.json.done` and appends the SPI pair, while failures become `.failed` siblings that operators can replay after fixing the root cause. `suite-cache/` holds the negotiated PRF/DH/PQC tuple for each IKE identifier, `suite-requests/` queues cache misses, and `child-suites/` stores the ESP snapshots emitted by `child_suite_cache.py`, so verifiers and validation scripts can recover the exact algorithms even after the tunnel is torn down. Outbound automation keeps its state in `tunnel-provisioning/`, where the provisioner records the legacy trigger, the authorised `gw-wan-out-*` profile, and the eventual result, preserving the intent that led to each initiation. Because `startup.sh` recreates the tree on every boot with root ownership and deterministic permissions, helpers never race for access, artefacts remain tied to a single compose run, and log shippers can mount the directory read-only from the host to collect evidence without perturbing the control plane.

4.5 Vendor notifier patch (0xA001)

4.5.1 Listener data structures and initialisation

The behaviour is baked into `src/libcharon/plugins/ext_auth/ext_auth_listener.c` via `multi-host-pep/pep-gateway/patches/ext-auth-runtime-cert.patch`. The patch extends the private state with a linked list of hints and a mutex so concurrent hook invocations cannot trample each other:

```
typedef struct {
    uint32_t unique_id;
    char *required_ke;
    bool pending_notify;
} hint_entry_t;

typedef struct private_ext_auth_listener_t {
    ...
    linked_list_t *hints;
    mutex_t *mutex;
} private_ext_auth_listener_t;
```

Helper routines such as `ensure_hint_locked()`, `store_required_ke()`, `set_pending_notify()`, and `clear_hint()` manage that list while holding the mutex, guaranteeing that every IKE_SA has at most one hint entry regardless of how many worker threads charon schedules. The listener constructor now initialises `hints` and `mutex`, wiring them into the public vtable so the new message hook can inspect outbound packets.

4.5.2 Environment enrichment and hint capture

Before launching `opa-auth-check.py` the listener now exports richer context: it serialises the negotiated PRF, DH group, and up to three PQ KEM identifiers into `PLUTO_IKE_*` variables, and it calls `acquire_peer_cert()` to drop the responder certificate (PEM plus a temporary file) into the environment. Once the Python script returns, the patch scans each line for `OPA_HINT`, which the helper emits whenever it denies a negotiation. The relevant fragment looks like:

```
char *hint = strstr(resp, "OPA_HINT");
if (hint && sscanf(hint,
```

```
        "OPA_HINT unique_id=%u required_ke=%31s",
        &hint_id, hint_level) >= 2)
{
    if (hint_id == unique_id)
    {
        store_required_ke(this, hint_id, hint_level);
    }
}
```

If the hook succeeded the listener calls `clear_hint()` to drop any cached guidance for that IKE_SA. If it failed, it flips the pending flag via `set_pending_notify(..., TRUE)` so the next outbound AUTH will carry a vendor notify.

4.5.3 Message hook and payload injection

The patch also registers a bespoke `message()` callback. For inbound IKE_AUTH requests it watches for notify `0xA001` and, if present, stores the peer's advertised requirement so the gateway can log it symmetrically. For outbound messages it injects the hint whenever the pending flag is set:

```
if (!incoming && type == IKE_AUTH)
{
    entry = find_hint(this, unique_id);
    if (entry && entry->pending_notify && entry->required_ke)
    {
        chunk_t payload = chunk_from_str(entry->required_ke);
        DBG1(DBG_CHD, "ext-auth: attaching required level '%s' for IKE_SA %u",
             entry->required_ke, unique_id);
        message->add_notify(message, FALSE,
                            NOTIFY_REQUIRED_LEVEL, payload);
        this->hints->remove(this->hints, entry, NULL);
        destroy_hint_entry(entry);
    }
}
```

The notify uses the vendor ID `0xA001` and the ASCII payload `required_ke=KE-L3;cert=SIG-L3-SUF`, which even legacy peers can parse without understanding PQ naming. Because hints are deleted as soon as the notify goes out (or when the IKE_SA dies), the list stays bounded during load tests.

4.5.4 Log signals and troubleshooting

Each stage leaves breadcrumbs.

`/var/log/ext-auth.log` records lines such as `OPA_HINT unique_id=37 required_ke=KE-L3 cert=SIG-L3-SUF reason=rekey_regression`. The listener mirrors every action with `DBG1` statements (`saved required level`, `scheduling notify`, `attaching required level`), so charon's syslog stream reveals exactly when the patch cached a hint or sent a vendor notify. If a partner supports the extension it will echo the same payload back when it retries; the inbound half of the message hook parses that response and logs `received required level 'KE-L3' for IKE_SA 37`, showing that both sides agree on the threshold. These signals are also forwarded to the decision logger: denials carry labels `required_ke_level` and `required_cert_level`, allowing Prometheus to alert on repeated downgrades, while packet captures show the binary notify, tying together wire data, policy rationale, and helper logs. Taken together, the patch turns opaque AUTH failures into actionable guidance without modifying partner templates or exposing internal policy services.

4.6 Observability and Monitoring

4.6.1 Decision-logger service

The `decision-logger` container is a hardened Express application (`decision-logger/server.js`) that ingests every structured event emitted by the helpers, stores it on disk, and exposes the derived metrics that Prometheus and Grafana consume. Security middleware loads first: `helmet`; restrictive CORS rules; HTTP parameter-pollution protection; JSON body size limits; and dual rate limiters (a general 3000 requests/min bucket plus a stricter `/logs` limiter) to prevent noisy helpers from overwhelming the logger, while a sanitizer strips control characters before they reach the filesystem. Every POSTed batch must satisfy the `decisionLogSchema` built with Joi; malformed entries are rejected with a 400 and never pollute the audit trail.

Once validated, the payload is passed through a Winston pipeline with three rotating transports: a JSON archive for raw records (`decisions-%DATE%.log`), an error-specific file, and a human-readable audit stream that renders synthetic lines such as '2024-03-18T10:22:53.184Z [7fc2ca5c] DECISION: DENY — REMOTE: 198.51.100.10 — SERVICE: payments — REASON: sig_level_insufficient'. Sensitive fields (passwords, API keys) are redacted by the `redactor()` format before the record ever hits disk. The same log is forwarded to stdout for quick inspection, so a failing helper can be diagnosed directly from `docker logs decision-logger`.

Counters live inside the `stats` structure: total decisions, allowdeny split, threat detections, fail-open fallbacks, per-level IKECHILD counts, rekey outcomes, and the distribution of AEADPFS primitives captured by `child_suite_cache.py`. Prometheus scrapes `/metrics` to obtain the exposed `opa_decision_logger_*` series (e.g., `opa_decision_logger_total_decisions`, `..._allowed_decisions`, `..._policy_decisions_by_type`, `..._rekey_operations_total`, `..._child_crypto_aead_total`/`..._child_crypto_pfs_total`), while `/health` and `/stats` expose JSON snapshots (uptime, last log time, decision rate) that the compose health checks use to confirm the service is alive before the gateway starts. Dashboard filters therefore operate on these outcome, and algorithm-based counters, not on per-message tags like `phase` or `reason`.

Ingress is tightly controlled. Only the gateway container is whitelisted in CORS, each request carries a UUID for traceability, and bulk uploads are limited to 1MB so an errant helper cannot wedge the process. In return the service provides a single, tamper-evident timeline that mirrors the control plane: synchronous decisions from `opa-auth-check.py`, asynchronous enforcement from `vici-child-manager.py`, provisioning attempts logged by `vici-tunnel-provisioner.py`, and selector verdicts from `updown_verifier.py`. All of them land under `/var/log/decision-logger.log` (bind-mounted to the host), giving the validation scripts and Grafana dashboards the same ground truth that operators would rely on in production.

4.6.2 Prometheus and Grafana wiring

Prometheus runs on the same `opa_network` bridge as the decision logger and scrapes only the logger's `/metrics` endpoint (no custom gateway exporter is defined in this compose; a node-exporter could be added separately if needed). The scrape jobs live in `prometheus/prometheus.yml` and use relabel rules to enforce TLS-free, bridge-local access so nothing on `external-net` can query those endpoints directly. Dashboards therefore work off the actual `opa_decision_logger_*` series, e.g. `opa_decision_logger_rekey_operations_totaloutcome="denied"` to spot make-before-break downgrades or `opa_decision_logge_child_validation_failures_total` for strict up/down hits, rather than per-message labels like phase/result/reason.

Grafana (under `grafana/provisioning/`) ships with prebuilt dashboards that overlay those metrics with extracts from the logs bind-mount. Panels show the end-to-end path for any identifier: the establishment verdict from OPA, the child manager's INIT/FAIL events, the verifier result, and the decision-logger latency histogram. Alert rules fire when suite extraction times out repeatedly, when the number of pending decisions exceeds a threshold (indicating a stuck helper), or when Prometheus stops seeing heartbeats from the CTI service. Because the compose stack bind-mounts `/var/log` from every container into the host's `logs/` directory, Grafana's Explore

view can pull the matching JSON snippet for any alert, giving operators the text they would otherwise have to read from the terminal. Together Prometheus and Grafana turn the structured events produced by the helpers into a live map of the control plane, complete with historical trends so regression tests can prove that a new patch maintains the desired failure posture.

4.6.3 Gateway logs and audit trail

Every helper keeps its own log under `/var/log/` inside the gateway container, and compose bind-mounts that directory back to the host so the validation scripts and dashboards see exactly what an operator would collect in production. `/var/log/ext-auth.log` is the authoritative trace of synchronous decisions: each invocation prints the unique identifier, peer metadata, suite source (cache, VICI, or charon), the OPA verdict with `required_ke_level`/`required_cert_level`, and any `OPA_HINT` lines that drive the vendor notifier. `/var/log/vici-suite-publisher.log` records every suite extraction attempt with latency and retry count, making it easy to spot VICI stalls or gaps in charon's log; `/var/log/vici-child-manager.log` mirrors the CREATE_CHILD_SA lifecycle (`INIT_REQUEST`, `INIT_SUCCESS`, `TEARDOWN`, retry backoffs) and lists the reqid/SPI pair so auditors can match it against `swanctl --list-sas`. Selector enforcement lives in `/var/log/opa-ext-auth/updown-verifier.log`, where each entry shows the CHILD name, negotiated selectors, expected selectors, and the action taken (`allow`, `teardown`). The outbound helpers add their own streams: `/var/log/vici-tunnel-provisioner.log` documents which legacy profile triggered the OPA decision, while `/var/log/child-suite-cache.log` links each CHILD to the AEAD/DH tuple captured for later replay.

Because all of these files sit under a single bind mount (`logs/` at the repository root), the automated test harness can bundle them into artefacts, Grafana's Explore view can display the surrounding context for any alert, and developers can diff runs to prove that a patch did not introduce spurious retries or silent downgrades. Combined with the decision logger's structured feed, these local logs complete the audit trail: a denial appears in ext-auth, the same identifier shows up in decision-logger with the policy rationale, the helper logs capture the retries or teardowns, and `swanctl --list-sas` (captured after each scenario) proves whether the kernel state matched the policy.

Chapter 5

Test

This chapter contains the description of several tests conducted on the implemented solution.

The first part outlines the capabilities provided by the implemented system, followed by a performance assessment examining execution time and the volume of packets and bytes exchanged during operation. Subsequently, a practical demonstration is presented to illustrate the DoS exposure inherent in enabling multiple unauthenticated multi-ADDKE exchanges during the initial IKE SA negotiation. A comparative analysis is then performed between a conventional legacy IKE configuration, where the CHILD SA is instantiated directly following the `IKE_AUTH` exchange, and the legacy childless procedure, which serves as the reference baseline against which our post-quantum establishment method is evaluated.

The evaluation further quantifies the overhead introduced by OPA, considering the cumulative duration attributed to *input parsing*, *query compilation*, *query evaluation*, and auxiliary processing within each PDP decision cycle.

Finally, a comprehensive analysis of the entire session-establishment workflow is provided. For every phase, we report the elapsed time, the OPA evaluation latency, the delay between the PDP decision and the subsequent CHILD SA installation, the child creation duration itself, and the number of packets, fragments and bytes exchanged. The additional cost of the ex-post CHILD validation, after which the secure tunnel is considered fully operational, is also included. This examination is repeated for all defined post-quantum security levels, enabling a detailed comparison of their respective performance characteristics.

5.1 Testbed

The experiments were conducted on an **HP Pavilion Gaming 15-`dk0000` Laptop PC** with the following specifications:

- **CPU**: Intel Core i5-9300H @ 2.40 GHz;
- **GPU**: NVIDIA GeForce GTX 1650 4 GB GDDR6 Dedicated VRAM;
- **RAM**: 8 GB DDR4;
- **Storage**: 256 GB SSD, 500 GB HDD;
- **OS**: Ubuntu 22.04 LTS, 64-bit.

5.2 Functional tests

Different tests have been executed to check the correctness of the features introduced.

In sequence, the section covers the establishment of the IKE SA, followed by the scenario in which the initial setup is rejected because the negotiated security level is insufficient, thereby activating the additional vendor-specific `Notifier`. Subsequently, the parameters of the installed child are further examined with ex-post validation to ensure that the installed child corresponds exactly to the configuration mandated by OPA for the relevant security level. Finally, the rekey downgrade validation mechanism is analysed.

5.2.1 IKE SA establishment

The first experiment illustrates the correct establishment flow of an IKE SA. The procedure begins with the `IKE_INIT` exchange, followed by an `IKE_INTERMEDIATE` round, which in this scenario is fragmented into two parts due to the sizeable ML-KEM − 1024 (Kyber_L5) key exchange payloads. Subsequently, the peers proceed with the `IKE_AUTH` exchanges, which themselves appear in multiple fragments in both directions. This fragmentation is primarily caused by the certificate transmission, which represents the dominant contributor to message size in this phase. In this testbed, the end-entity certificate relies on ML-DSA − 65 (Dilithium_3) for its public key and issuer's signature algorithms.

```
[IKE] initiating IKE_SA banka-responder[4] to 203.0.113.2
[ENC] generating IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(FRAG_SUP) N(HASH_ALG) N(REDIR_SUP) N(IKE_INT_SUP) V ]
[NET] sending packet: from 198.51.100.10[500] to 203.0.113.2[500] (384 bytes)
[NET] received packet: from 203.0.113.2[500] to 198.51.100.10[500] (461 bytes)
[ENC] parsed IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) CERTREQ N(FRAG_SUP) N(HASH_ALG) N(CHDLESS_SUP) N(IKE_INT_SUP) N(MULT_AUTH) V ]
[IKE] received strongSwan vendor ID
[CFG] selected proposal: IKE:AES_GCM_16_256/PRF_HMAC_SHA2_512/ECP_521/KE1_KYBER_L5
[IKE] remote host is behind NAT
[IKE] received cert request for "CN=PQ-Gateway Root CA"
[IKE] received cert request for "O=PQ-Gateway Lab, CN=PQ-Gateway IKE CA PQ"
[IKE] received cert request for "O=PQ-Gateway Lab, CN=PQ-Gateway Root CA PQ"
[IKE] received cert request for "CN=Post-Quantum CA, OU=strongSwan, O=Linux strongSwan"
[ENC] generating IKE_INTERMEDIATE request 1 [ KE ]
[ENC] splitting IKE message (1633 bytes) into 2 fragments
[ENC] generating IKE_INTERMEDIATE request 1 [ EF(1/2) ]
[ENC] generating IKE_INTERMEDIATE request 1 [ EF(2/2) ]
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (450 bytes)
[NET] received packet: from 203.0.113.2[4500] to 198.51.100.10[4500] (1248 bytes)
[ENC] parsed IKE_INTERMEDIATE response 1 [ EF(1/2) ]
[ENC] received fragment #1 of 2, waiting for complete IKE message
[NET] received packet: from 203.0.113.2[4500] to 198.51.100.10[4500] (450 bytes)
[ENC] parsed IKE_INTERMEDIATE response 1 [ EF(2/2) ]
[ENC] received fragment #2 of 2, reassembled fragmented IKE message (1633 bytes)
[ENC] parsed IKE_INTERMEDIATE response 1 [ KE ]
[IKE] sending cert request for "O=PQ-Gateway Lab, CN=PQ-Gateway Root CA PQ"
[IKE] authentication of 'banka' (myself) with DILITHIUM_3 successful
[IKE] sending end entity cert "CN=banka"
[ENC] generating IKE_AUTH request 2 [ IDi CERT N(INIT_CONTACT) CERTREQ IDr AUTH N(MOBIKE_SUP) N(NO_ADD_ADDR) N(MULT_AUTH) N(EAP_ONLY) N(MSG_ID_SYN_SUP) ]
[ENC] splitting IKE message (9056 bytes) into 8 fragments
[ENC] generating IKE_AUTH request 2 [ EF(1/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(2/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(3/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(4/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(5/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(6/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(7/8) ]
[ENC] generating IKE_AUTH request 2 [ EF(8/8) ]
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (751 bytes)
[IKE] retransmit 1 of request with message ID 2
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
[NET] sending packet: from 198.51.100.10[4500] to 203.0.113.2[4500] (1248 bytes)
```

Figure 5.1. Initial IKE SA exchanges

After authentication concludes, but before the IKE SA is fully installed, the `ext-auth` plugin intercepts the workflow. At this point, the state machine is deliberately paused while the plugin's script extracts all relevant metadata, including the peer identity, source address, negotiated cryptographic suite and certificate attributes. These are transmitted to the PDP (OPA) for policy evaluation.

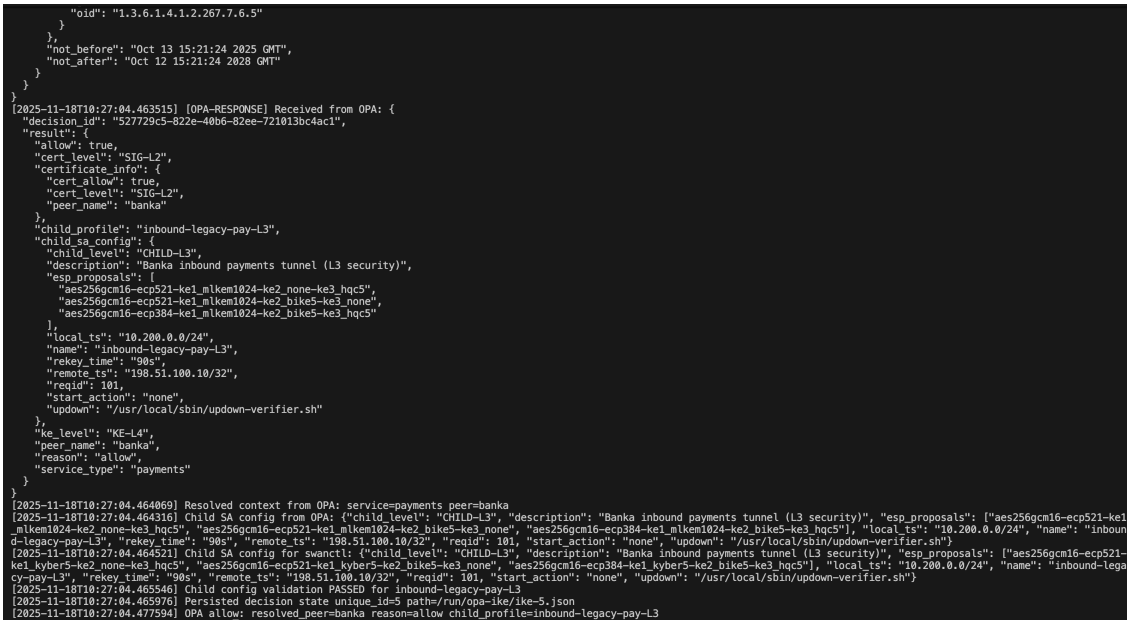

Figure 5.2. IKE SA metadata collection via `ext-auth`

Based on this input, the PDP returns a decision that includes both the approved security level and the complete child SA configuration template to be enforced. Before finalising the installation of the IKE SA, the returned template is validated against the locally defined `swanctl` configuration to ensure that the child to be installed corresponds exactly to the policy mandated one. Once this final consistency check succeeds, the `ext-auth` plugin issues an authorisation success verdict and the IKE SA is installed.

```
[ENC] received fragment #13 of 13, reassembled fragmented IKE message (14593 bytes)
[ENC] parsed IKE_AUTH response 2 [ IDr CERT CERT AUTH N(MOBIKE_SUP) N(ADD_4_ADDR) N(ADD_4_ADDR) ]
[IKE] received end entity cert "CN=pep-gateway"
[IKE] received issuer cert "O=PQ-Gateway Lab, CN=PQ-Gateway IKE CA PQ"
[CFG]   using trusted certificate "CN=pep-gateway"
[CFG]   using trusted intermediate ca certificate "O=PQ-Gateway Lab, CN=PQ-Gateway IKE CA PQ"
[CFG]   using trusted ca certificate "O=PQ-Gateway Lab, CN=PQ-Gateway Root CA PQ"
[CFG]   reached self-signed root ca with a path length of 1
[IKE] authentication of 'pep-gateway' with DILITHIUM_3 successful
[IKE] peer supports MOBIKE
[IKE] IKE_SA banka-responder[4] established between 198.51.100.10[banka]...203.0.113.2[pep-gateway]
[IKE] scheduling rekeying in 112s
[IKE] maximum IKE_SA lifetime 124s
initiate completed successfully
```

Figure 5.3. IKE SA successfully installed

5.2.2 IKE SA establishment denial due to insufficient security level

When the PDP rejects an incoming IKE SA request because the negotiated parameters do not satisfy the minimum security requirements, the gateway halts the establishment process. In this case, the PDP response includes both the reason for the denial and the minimum KE and signature levels expected for that specific peer. Consequently, the responder returns a failure during the `IKE_AUTH` exchange: besides the standard `AUTH_FAILED` notification, the gateway also emits a vendor-specific notification whose payload explicitly encodes the required KE and signature levels.

```
[NET] sending packet: from 198.51.100.11[4500] to 203.0.113.2[4500] (760 bytes)
[NET] received packet: from 203.0.113.2[4500] to 198.51.100.11[4500] (90 bytes)
[ENC] parsed IKE_AUTH response 2 [ N(AUTH_FAILED) N(BEET_MODE) ]
[IKE] received AUTHENTICATION_FAILED notify error
initiate failed: establishing IKE_SA 'partnerb-responder' failed
```

Figure 5.4. IKE SA establishment denied owing to an insufficient security level

The remote peer, which also includes the corresponding parsing patch, is capable of recognising, decoding, and logging this additional notification. As illustrated in fig. 5.5, the peer processes the vendor-specific payload and records the required cryptographic levels, thereby enabling diagnostic feedback and future renegotiations that comply with the advertised security policy.

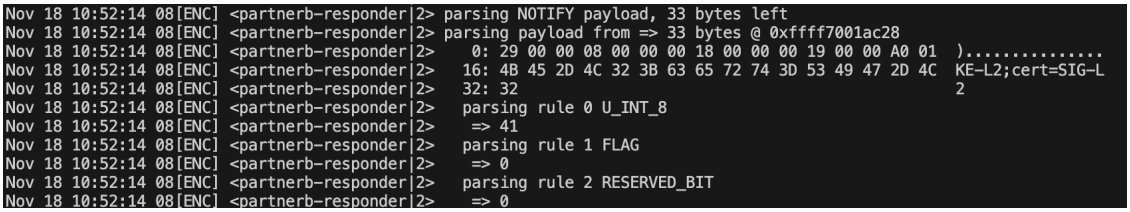

Figure 5.5. Peer-side parsing of vendor-specific notification indicating required security levels

5.2.3 Child SA installation

As already noted, in the final phase of the exchange, once the PDP returns the authorised child template, the gateway verifies that this configuration exactly matches the corresponding entry in `swanctl.conf`. If the template is consistent with the locally declared policy, it is persisted under `/run/opa-ike/ike-<id>.json`, thereby recording both the decision and the full set of parameters required for subsequent enforcement.

```
docker exec pep-gateway cat /run/opa-ike/ike-2.json.done

{
  "schema_version": 3,
  "created_by": "opa-auth-check",
  "timestamp": 1763461322.9229274,
  "ike_unique_id": "2",
  "role": "initiator",
  "service": "payments",
  "service_hint": "payments",
  "allow": true,
  "reason": "allow",
  "required_ke_level": "unknown",
  "required_cert_level": "unknown",
  "child_profile": "to-banka-L3",
  "child_sa_config": {
    "child_level": "CHILD-L3",
    "description": "BankA outbound payments tunnel (L3)",
    "esp_proposals": [
      "aes256gcm16-ecp521-ke1_kyber5-ke2_none-ke3_hqc5",
      "aes256gcm16-ecp521-ke1_kyber5-ke2_bike5-ke3_none",
      "aes256gcm16-ecp384-ke1_kyber5-ke2_bike5-ke3_hqc5"
    ],
    "local_ts": "10.200.0.0/24",
    "name": "to-banka-L3",
    "rekey_time": "90s",
    "remote_ts": "198.51.100.10/32",
    "reqid": 201,
    "start_action": "none",
    "updown": "/usr/local/sbin/updown-verifier.sh"
  },
  "child_sa_config_opa": {
    "child_level": "CHILD-L3",
    "description": "BankA outbound payments tunnel (L3)",
    "esp_proposals": [
      "aes256gcm16-ecp521-ke1_mlkem1024-ke2_none-ke3_hqc5",
      "aes256gcm16-ecp521-ke1_mlkem1024-ke2_bike5-ke3_none",
      "aes256gcm16-ecp384-ke1_mlkem1024-ke2_bike5-ke3_hqc5"
    ],
    "local_ts": "10.200.0.0/24",
    "name": "to-banka-L3",
    "rekey_time": "90s",
    "remote_ts": "198.51.100.10/32",
    "reqid": 201,
    "start_action": "none",
    "updown": "/usr/local/sbin/updown-verifier.sh"
  },
  "peer": {
```

Figure 5.6. Persisted OPA decision and child template

From this point, the `vici-child-manager` retrieves the approved child profile name from the stored decision file and triggers its installation. The ensuing CHILD SA negotiation follows the standard strongSwan flow. However, in this instance the level-4 child requires multiple fragments, reflecting the computational weight and message size associated with the HQC-5 component of the multi-KEM ESP proposal.

Figure 5.7. Authorised CHILD SA being initiated via `VICI`

Once installed, the CHILD SA is formally associated with the pre-existing corresponding IKE SA, completing the childless workflow in which the child is created only after the IKE SA has been authorised and validated.

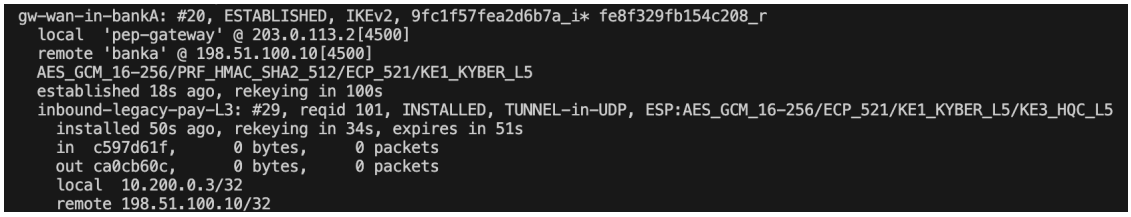

Figure 5.8. Final CHILD SA bound to the IKE SA after installation

5.2.4 Child ex-post validation

After the CHILD SA has been successfully installed, the `updown` verification hook performs an ex-post consistency check, consulting again the PDP. Using the parameters obtained via VICI, it inspects the effectively negotiated ESP suite and traffic selectors, ensuring that they precisely correspond to the configuration authorised by the PDP for that specific connection. Any deviation, whether a weaker cryptographic proposal, an unexpected selector, or an omitted KEM component, would trigger a controlled teardown of the tunnel to prevent the establishment of a non-compliant security association.

Once the verification stage completes, the outcome is forwarded to the *decision logger*, which records both the authoritative OPA mandate and the parameters of the CHILD SA as actually instantiated on the gateway. This produces an auditable, time-stamped record of the enforcement pipeline, capturing the achieved level, and the result of the ex-post validation.

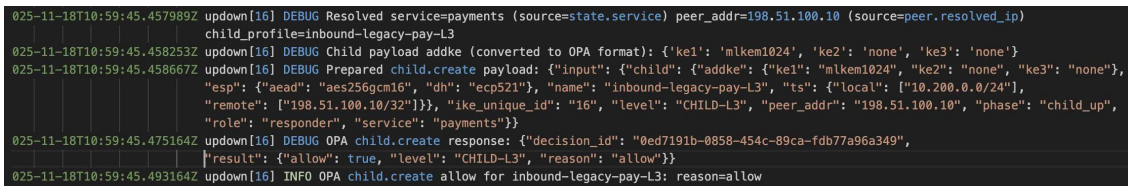

Figure 5.9. Ex-post validation performed by the `updown` hook

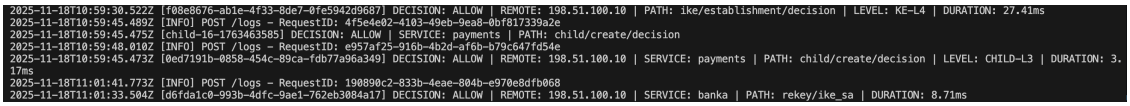

Figure 5.10. Decision Logger entry for validated CHILD SA

Selector mismatch enforcement

To validate that the ex-post validation truly prevents non-compliant CHILD SAs, we crafted a tampering scenario using the `tamper-updown.sh` script. The test deliberately poisons the cached metadata used by the `updown` hook so that the traffic selectors authorised by the PDP differ from those negotiated on the wire.

What the script does

- Restarts `gateway` and `banka`, reloads `swanctl` and establishes the baseline IKE/CHILD (`gw-wan-out-bankA` / `to-banka-L3`).
- Suspends the helper processes (`vici-suite-publisher`, `child_suite_cache`, `vici-child-manager`) to freeze metadata updates.
- Locates the cached child metadata (`/run/opa-ike/child-suites/ike-*-to-banka-L3.json`) and tampers it: `ts.local` is forced to `10.200.0.0/24` while the on-wire child negotiation proposes `10.200.0.3/32`.
- Creates a fail-close flag (`/run/opa-ike/fail_on_mismatch`) so the `updown` hook will tear down the CHILD on any mismatch.
- Forces a fresh IKE/CHILD negotiation (terminate and re-initiate) so that `updown` reads the tampered metadata during validation.
- Removes the flag and resumes the helper processes after the test.

Expected behaviour When the negotiated selectors differ from the cached OPA decision, the `updown` hook, running in fail-close mode, must immediately terminate the CHILD SA and log the mismatch, rather than allowing an out-of-policy tunnel to persist.

Observed result The log in fig. 5.11 shows cache hits for the tampered metadata, a clear diff between `meta_local/meta_remote` and `config_local/config_remote`, and the hook reacting with an immediate teardown:

- `Child config TS differ from metadata (meta_local=['10.200.0.0/24'] ... config_local = ['10.200.0.3/32'])`
- `ERROR Selector mismatch detected and UPDOWN_FAIL_ON_MISMATCH enabled; terminating child immediately`
- `WARN Terminating child to-banka-L3 due to policy deny`
- `INFO action=down ...`

Each attempt to establish the child in this poisoned state is denied and torn down, demonstrating that any deviation from the PDP-approved selectors is blocked at install time.

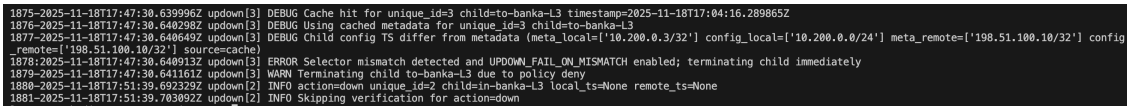

Figure 5.11. Updown hook detecting selector mismatch (OPA metadata vs negotiated TS) and tearing down the CHILD SA

5.2.5 Rekey validation gate

A final validation stage is executed during the lifetime of an established IKE SA in order to prevent any form of cryptographic downgrade when rekeying occurs. When strongSwan signals that an IKE rekey is imminent, the `vici-child-manager` intercepts the event and leverages the native *make-before-break* mechanism to retrieve both the "old" and the "new" cryptographic suites associated with the rekey attempt. The suites, are packaged and submitted to OPA through the dedicated rekey policy endpoint.

```
2025-11-18 12:14:42,626 - INFO - IKE REKEY detected for gw-wan-out-bankA: uniqueid 9 -> 10
2025-11-18 12:14:42,626 - INFO - Preparing OPA rekey validation for gw-wan-out-bankA
2025-11-18 12:14:42,626 - INFO -   Old suite: {'aead': 'aes256gcm16', 'prf': 'sha512', 'dh': 'ecp521',
                         'addke': {'ke1': 'mlkem1024', 'ke2': 'none', 'ke3': 'none'}}
2025-11-18 12:14:42,626 - INFO -   New suite: {'aead': 'aes256gcm16', 'prf': 'sha512', 'dh': 'ecp521',
                         'addke': {'ke1': 'mlkem1024', 'ke2': 'none', 'ke3': 'none'}}
2025-11-18 12:14:42,627 - INFO - Calling OPA rekey policy: http://opa-server:8181/v1/data/rekey/ike_sa
2025-11-18 12:14:42,627 - INFO - Payload: {
  "input": {
    "service": "banka",
    "connection_name": "gw-wan-out-bankA",
    "peer_addr": "198.51.100.10",
    "old": {
      "aead": "aes256gcm16",
      "prf": "sha512",
      "dh": "ecp521",
      "addke": {
        "ke1": "mlkem1024",
        "ke2": "none",
        "ke3": "none"
      }
    },
    "new": {
      "aead": "aes256gcm16",
      "prf": "sha512",
      "dh": "ecp521",
      "addke": {
        "ke1": "mlkem1024",
        "ke2": "none",
        "ke3": "none"
      }
    }
  }
}
2025-11-18 12:14:42,644 - INFO - OPA rekey decision: allow=True, reason=allow, action=promote_new_delete_old
2025-11-18 12:14:42,645 - INFO - IKE rekey ALLOWED by OPA for gw-wan-out-bankA
```

Figure 5.12. Rekey request intercepted and evaluated against PDP policy

The PDP evaluates whether the proposed replacement satisfies the minimum security requirements for the associated service. If the new suite preserves or improves the previously achieved level, the `vici-child-manager` promotes the fresh IKE SA and subsequently tears down the old instance. Conversely, if the rekey attempt results in a weaker configuration, the new IKE SA is rejected and the existing one is retained, thus preventing any inadvertent or malicious downgrade.

```
2025-11-18T12:14:44.125Z [INFO] POST /logs - RequestID: 73c82307-f79b-4794-b4c5-29a74e91406a
2025-11-18T12:14:42.638Z [701bef00-7fc3-4f81-b57c-fe03793111a5] DECISION: ALLOW | REMOTE: 198.51.100.10 | SERVICE: banka | PATH: rekey/ike_sa | DURATION:
3.29ms
```

Figure 5.13. Audit entry showing the validated rekey decision and resulting action

5.3 Performance tests

In this section, we present the performance evaluation of the proposed solution and contrast it with a classical baseline composed of a standard IKE exchange followed by immediate CHILD SA

creation. The analysis examines the latency introduced throughout the entire establishment workflow, the additional computational effort required for post-quantum primitives, and the resulting increase in packet count, fragmentation and byte overhead.

Furthermore, the evaluation isolates the delay attributable to PDP consultation at each stage of the negotiation process, from the initial policy query to the moment at which a functional protected channel becomes available. A detailed breakdown of the policy evaluation pipeline is also provided, highlighting the contribution of each component, parsing, compilation, rule matching and final decision synthesis, to the overall OPA processing time observed during the establishment procedure.

5.3.1 Unauthenticated multi-KEM overhead in IKE establishment

This experiment quantifies the cost of adding multiple KEMs during the unauthenticated IKE SA setup, highlighting the DoS exposure that motivates the use of a single KEM in the IKE establishment and the use of the childless design adopted in this work. Two runs are compared: a baseline configuration in which the gateway negotiates a single post-quantum KE (ML-KEM-1024) and a multi-KEM configuration that adds BIKE-5 and HQC-5 in ADDKE #2 and ADDKE #3, respectively. Even though only three out of the seven additional KEM slots envisaged by RFC-9370 are exercised, the overall exchange time more than doubles (from roughly 100 ms to 217 ms), and the total UDP volume and packet count increase by factors of approximately 2.16 and 2.11, as illustrated in figs. 5.14 to 5.16. The bulk of the extra cost stems from the HQC exchange, which alone accounts for over 40 % of the total duration in the multi-KEM case.

The measurements are obtained by capturing the full IKE traffic on the `gateway` container using the helper script:

```
&& ./scripts/capture-and-analyze-ike.sh \
     pep-gateway gw-wan-out-bankA 198.51.100.2
```

The script first terminates any existing IKE SA, then starts `tcpdump` inside the container to record packets on UDP ports 500/4500. The resulting PCAP is copied to the host and analysed by `analyze-pcap-timing.py`, which invokes `tshark` to extract timestamps, message types and lengths, groups packets into phases (`IKE_SA_INIT`, `IKE_INTERMEDIATE` #1–#3, `IKE_AUTH`), and computes, for each phase, the duration, number of packets and bytes sent/received at the UDP level.

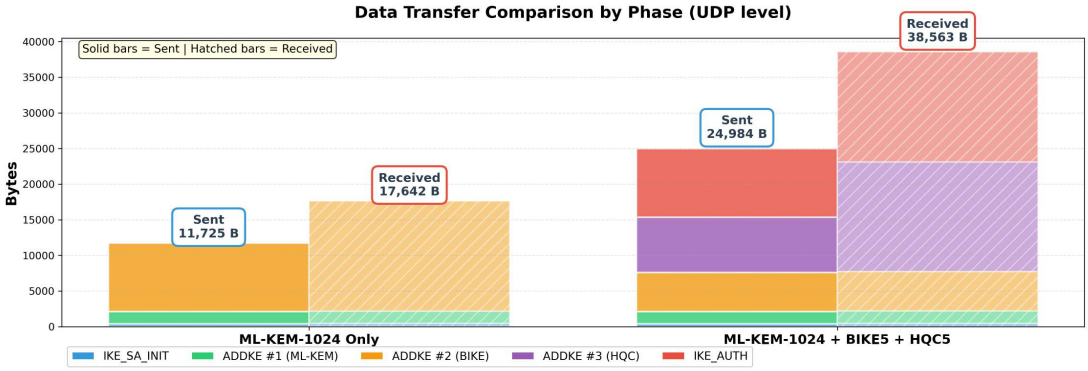

Figure 5.14. Comparison of transmitted and received data volumes for single-KEM and multi-KEM IKE establishment

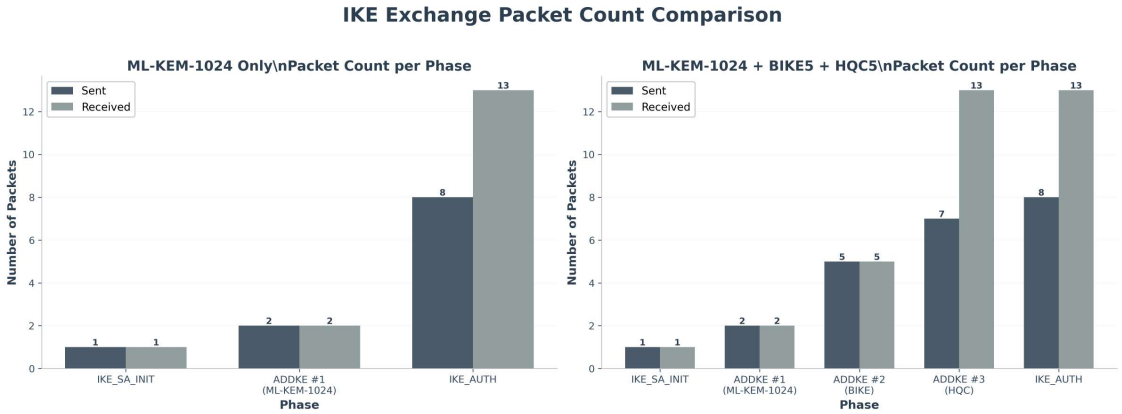

Figure 5.15. Packet count distribution across IKE phases for ML-KEM only versus multi-KEM setups

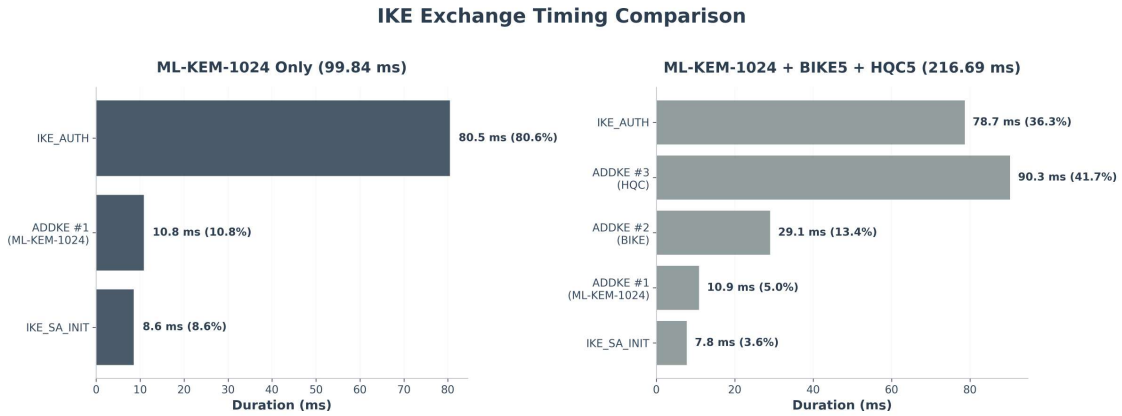

Figure 5.16. Phase-level timing comparison between ML-KEM only and multi-KEM IKE negotiations

5.3.2 Legacy IKE establishment: normal versus childless mode

A second experiment contrasts a conventional legacy IKE setup, in which a CHILD SA is created immediately during `IKE_AUTH`, with a childless initial exchange where only the IKE SA is established. As specified by RFC-7296, in the normal case the first CHILD SA is derived directly from the key material negotiated in `IKE_SA_INIT`. The `IKE_AUTH` message carries, in addition to the authentication payloads, also the ESP proposal and traffic selectors for that child. While additional effective independent key exchanges are used solely for subsequent `CREATE_CHILD_SA` operations to provide PFS.

The experiment is automated by the `simple-ike-comparison.sh` script, which alternates between the normal and childless configurations of the `gateway` and `host-legacy` containers. For each mode the script restarts the containers, reloads the `swanctl` configuration, captures the full handshake on the gateway with `tcpdump`, and derive the total establishment time and the number of bytes observed on the wire. The two JSON reports are finally compared, which computes the incremental overhead introduced by negotiating a CHILD SA within `IKE_AUTH`.

The results, summarised in fig. 5.17, show that embedding a CHILD SA directly within the IKE handshake introduces a measurable increase in both data transfer and processing time. From a bandwidth perspective, the handshake grows by approximately 7 %, a consequence of the additional payloads carried in the final IKE_AUTH response. Specifically, the response size increases by roughly 80 B, consisting of the ESP `SA` payload, the `TSi` and `TSr` payloads, and a small amount of padding and alignment overhead.

On the request path, the overhead is more pronounced (476 B), as the initiator must transmit the full CHILD SA proposal and the associated traffic selectors, and additional notify payloads. In our measurements, this request is split across multiple IP-level fragments, which increases the number of packets on the wire and the per-packet processing overhead on both peers.

In terms of latency, the impact is more substantial: the full establishment time more than triples, increasing from approximately 157 ms in the childless case to around 539 ms when the CHILD SA is negotiated in-band. This slowdown arises almost entirely from the additional processing and fragmentation required to negotiate the child proposal during IKE_AUTH.

These results show the security hardening ensured by our architecture, in which the gateway deliberately adopts the childless mode for the initial IKE SA, and subsequently creates a fresh CHILD SA with dedicated post-quantum key exchanges once the PDP has authorised the connection. This separates the tunnel keys from the initial IKE negotiation, keeps the IKE proposal itself lightweight (single standardised KEM), and shifts the additional post-quantum cost into an explicitly authorised and tightly controlled CHILD establishment step, assuring a comprehensive supplementary exchange even for the initial child creation.

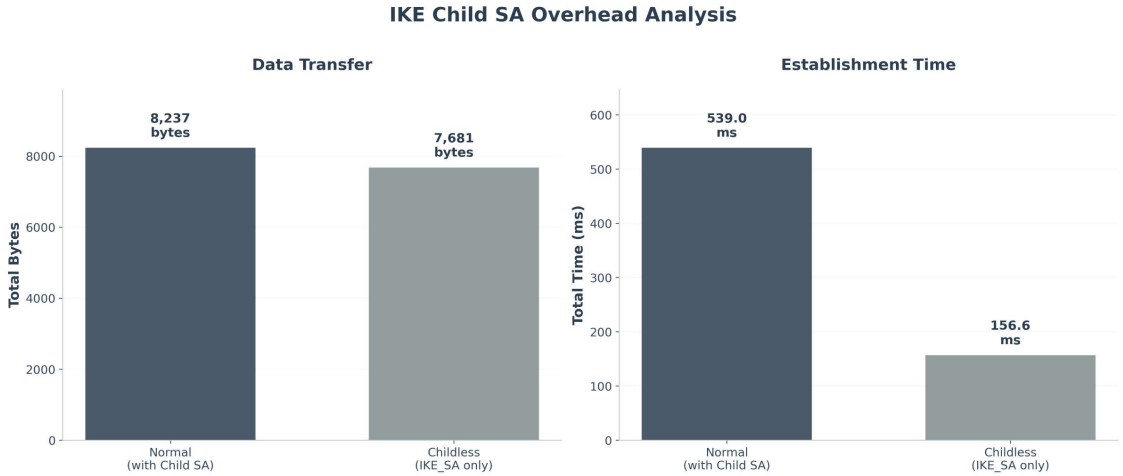

Figure 5.17. Data and time overhead of a normal IKE establishment with CHILD SA versus childless mode

5.3.3 Policy evaluation timing analysis

To quantify the computational cost introduced by the policy-driven control plane, we analyse the evaluation time of each OPA rule involved in the security workflow: the `ike/establishment` decision, the `child/create` decision, and the `rekey/ike_sa` downgrade prevention gate. Figure 5.18 reports the breakdown of the average latency observed for each policy phase, distinguishing the time spent parsing the input, compiling the Rego query, evaluating the policy logic, and the residual runtime overhead.

The evaluation of the IKE establishment policy is, by design, the most time-consuming stage. This rule constitutes the core of the PDP's security assessment and therefore performs the most extensive set of checks. Specifically, the decision must:

- authenticate and classify the remote peer identity;
- determine whether the requested connection is admissible given the declared policy set;
- assess whether the negotiated IKE security level satisfies the minimum requirements for that peer or service;
- validate the remote certificate, including level requirements and subject matching;
- derive the authorised child template to be enforced by the gateway.

For these reasons, its cost dominates the evaluation process, averaging 10.48 ms, with more than 92 % of this time attributed to actual query evaluation.

By contrast, the subsequent phases, `child/create` and `rekey/ike_sa`, are structurally lighter. Both rules primarily act as consistency checks that resemble pattern-matching over previously authorised state. For child creation, the rule requires substantially fewer logical operations and averages 4.15 ms.

Similarly, the rekey rule, as this logic is predominantly a structural comparison, it exhibits the smallest evaluation cost, averaging 3.70 ms across runs.

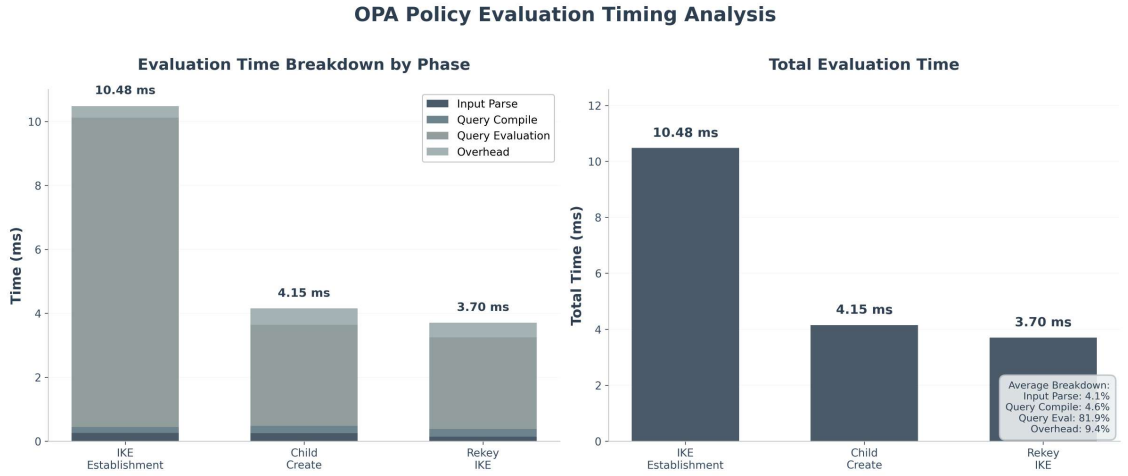

Figure 5.18. Breakdown of OPA evaluation times across the IKE establishment, child creation and rekey security gates

5.3.4 Comprehensive evaluation across security levels

This section consolidates the multi-level performance analysis by examining, in a unified manner, the timing characteristics, network overhead, and fragmentation behaviour observed across the four security levels. The measurements derive from controlled experiments conducted through the benchmarking framework executed via the `run_benchmark_suite.sh` script. Packets are captured on the `gateway` with `tcpdump` into PCAP traces, then parsed with `tshark` and correlated with `vici` and OPA audit logs to extract timing and traffic metrics. Although, it is important to highlight that the timing values exhibit a degree of non-determinism. Such variability primarily stems from the packet capture methodology, the behaviour of the Docker-based execution environment, and short-lived network congestion during the exchange phases. Consequently, the collected timings should be regarded as representative rather than absolute. It is worth noting that, across all evaluated configurations, the required signature strength is fixed at level 2. This choice provides a satisfactory level of assurance for authentication while avoiding the heavier level 3 option, whose cost is dominated by certificate exchange. At the same time, level 2 remains sufficient to study the differences between security levels arising from the distinct Additional Key Exchanges (ADDKEs) used in each case. However, this decision is easily reversible: since the security level of the KE is decoupled from that of the signature, it is sufficient to adjust the signature level in `service_classes.rego` for the relevant external service, and the new requirement will be propagated automatically during the validation phase.

Traffic, bytes and fragmentation

The aggregate network statistics in figs. 5.19 and 5.20 reflect the intrinsic cost of increasing post-quantum security levels. Packet counts, transferred bytes and the number of IKE fragments all grow consistently from L1 through L4. This trend is especially visible for the child installation stage, where the cryptographic workload directly influences the payload size and fragmentation

ratio. At level 4, where ML-KEM-1024, BIKE-5, and HQC-5 are jointly negotiated, the child establishment accounts for the highest number of fragments (34 fragments in total), and a corresponding traffic volume of approximately 37.1 KiB. Such growth is entirely expected as larger post-quantum public keys and ciphertexts must be transmitted during the ESP negotiation.

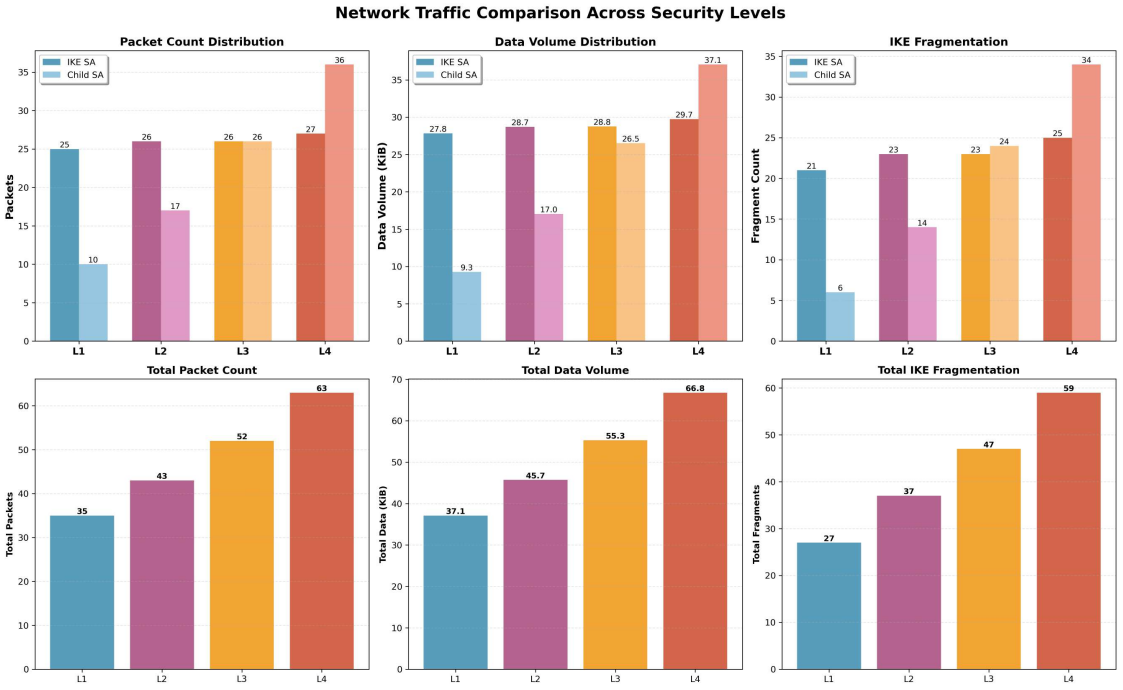

Figure 5.19. Network traffic comparison across all security levels

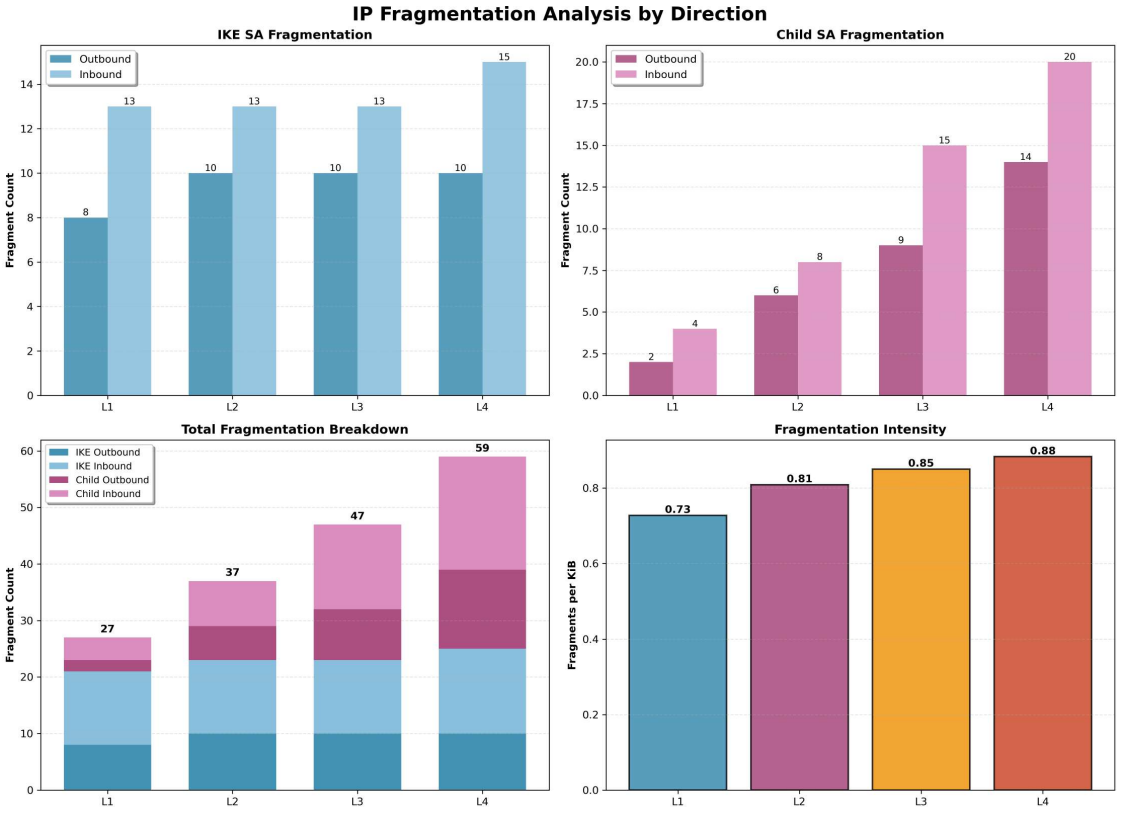

Figure 5.20. IP fragmentation comparison across security levels

Macroscopic timing behaviour

Although the cryptographic payload grows with the level, the global timing bottlenecks remain consistent across all tiers. The most significant contributor to the total duration is the *IKE authentication phase*, dominated by certificate exchange and signature verification. A second non-negligible component is the delay between the PDP's decision and the start of child negotiation, which varies according to system load, scheduling of the `vici-child-manager`, and transient conditions within the Docker network stack.

The child negotiation time (purple bar in fig. 5.21) increases monotonically with the security level, as anticipated, and reaches its maximum at level 4. This reflects the cumulative cost of negotiating three post-quantum KEMs sequentially. The evaluation also accounts for the ex-post validation performed by the `updown` mechanism to ensure that the installed suites strictly match the authorised template, to consider complete the entire secure tunnel establishment.

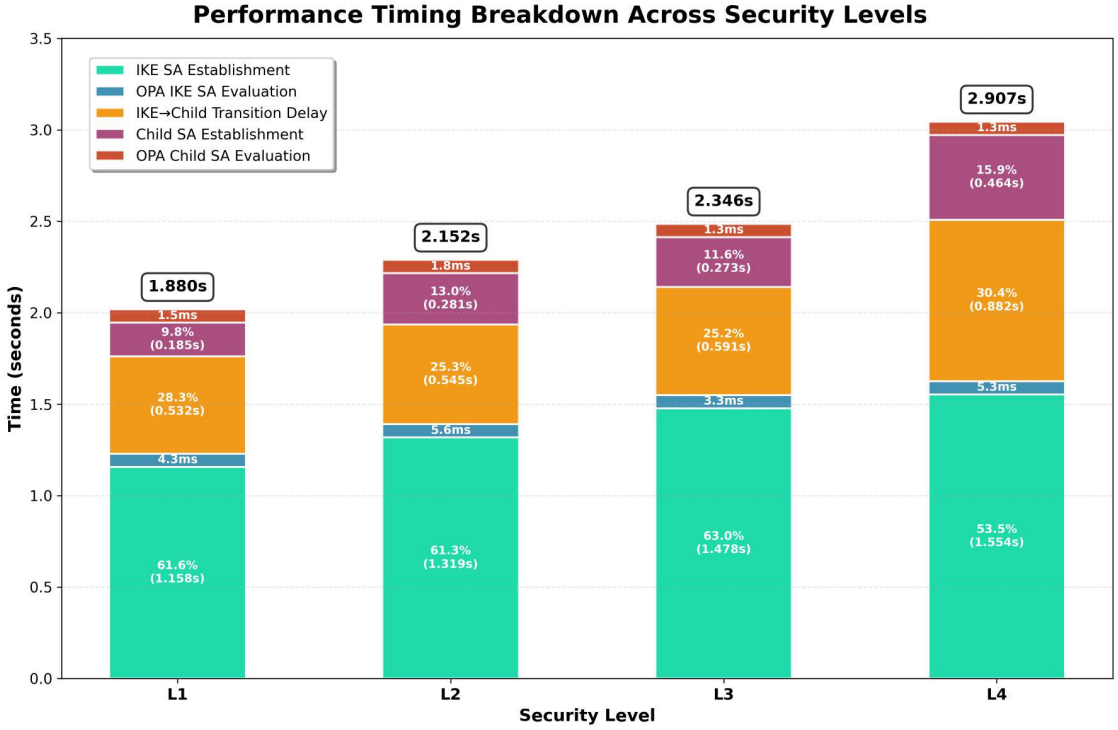

Figure 5.21. Performance timing breakdown across security levels

Overall interpretation

Despite a clear increase in establishment time relative to the classical legacy scenario, where the first child is derived immediately after `IKE_AUTH` with negligible additional negotiation, the proposed solution offers a qualitatively different security posture. The policy-driven approach enables complete control over the negotiation pipeline: the PDP can enforce the precise KE, signature and child-level combinations required for each service, while decoupling the establishment of the transmission tunnel from the IKE SA itself. The resulting overhead represents a one-time cost incurred only during the initial establishment, and is justified by the stringent security guarantees needed in the target deployment environment.

Chapter 6

Conclusion

This document has introduced and evaluated a policy-centric, post-quantum aware IPsec gateway. The proposed system is designed to shield legacy infrastructures that cannot yet adopt post-quantum schemes natively, while still allowing them to benefit from quantum-resistant protections. Informed by the analysis of the quantum threat landscape, the NIST PQC standardisation process and the practical obstacles of migrating IPsec and IKEv2, the gateway operates as a translation point: it terminates conventional IKEv2 sessions with a legacy LAN and reestablishes connectivity towards post-quantum capable peers on an external network, enforcing centrally defined cryptographic policies and target security levels along the way.

A central premise of this work is that *cryptographic agility*, rather than the selection of one specific algorithm, is the main design requirement for systems expected to face both classical and quantum-capable adversaries over time. To this end, the gateway does not embed fixed cryptographic choices in its data plane. Instead, it follows a policy-driven architecture in which an external Policy Decision Point (PDP), realised with Open Policy Agent (OPA), determines admissible algorithms, key exchange patterns, signature schemes and per-service security levels. The strongSwan-based gateway acts as a Policy Enforcement Point (PEP), terminating tunnels on both sides and consulting the PDP, before allowing security relevant state changes. This clean separation of roles decouples policy evolution from packet processing, so that changes in security posture can be realised by updating policy modules and data bundles, without modifying or restarting the enforcement component.

The prototype has been validated through functional tests that exercise the full negotiation workflow. The experiments show that the gateway correctly intercepts authenticated Internet Key Exchange (IKE) exchanges, submits a rich context to the PDP, and proceeds only when an internally consistent child configuration is authorised. When a proposed cryptographic suite fails to meet the minimum requirements, the gateway rejects the IKE association and returns a diagnostic notification that can be logged and interpreted by the peer. Child SA installation is shown to derive exclusively from OPA-approved templates, which are recorded and cross-checked against the actually negotiated traffic selectors and ESP transforms. Any deviations cause a fail-closed teardown. Rekey validation further confirms that downgrade attempts are systematically blocked, and that new IKE SAs are accepted only when they preserve or raise the previously achieved security level.

Furthermore, the performance evaluation provides a quantitative perspective on the costs of adopting post-quantum primitives and policy-driven control. Measurements show that performing multiple unauthenticated ADDKE rounds during IKE SA setup substantially increases handshake latency and traffic volume, and makes the protocol more exposed to denial-of-service. This observation justifies constraining IKE itself to a single standardised KEM while moving additional post-quantum work into an authenticated, policy-gated child establishment phase. Additional benchmarks quantify the contribution of the PDP: even for the most complex rule, which performs peer classification, admissibility checks, level verification, certificate validation and child-template derivation, the average evaluation time remains in the order of a few tens of milliseconds, with lighter rules incurring only a few milliseconds. Across four configured security levels, packet

counts, bytes and fragmentation grow as larger post-quantum keys and ciphertexts are used, yet the overall timing profile remains dominated by certificate handling and the actual delay from the OPA response to the mandated child initialisation, with PDP evaluations contributing only a modest fraction. These results suggest that a policy-driven, post-quantum IPsec gateway is practically deployable, with overheads that are visible but acceptable for many site-to-site and high-value service scenarios.

This work demonstrates comprehensively that it is both technically and operationally achievable to build a post-quantum IPsec gateway that combines crypto-agility, fine-grained policy control and rich observability, while still maintaining interoperability with existing IKEv2 deployments. At the same time, the evaluation has been limited to a controlled laboratory setting. Real-world deployments will need to consider a broader range of hardware platforms, traffic profiles, failure modes, long-term data-plane performance and highly available, horizontally scalable PDP designs. Addressing these aspects, and generalising the policy-centric pattern beyond IPsec, for example to post-quantum aware TLS termination, service meshes and application-layer gateways, defines a natural agenda for future work and a coherent route towards quantum-safe, crypto-agile network security at scale.

Bibliography

[1] European Union Agency for Cybersecurity (ENISA), "Post-Quantum Cryptography: Current state and quantum mitigation." ENISA Report, December 2021, https://www.enisa.europa.eu/publications/post-quantum-cryptography-current-state-and-quantum-mitigation

[2] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on Post-Quantum Cryptography." NIST Internal Report 8105, April 2016, DOI 10.6028/NIST.IR.8105

[3] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", 35th Annual Symposium on Foundations of Computer Science, Santa Fe (NM, USA), Nov 20-22, 1994, pp. 124–134, DOI 10.1109/SFCS.1994.365700

[4] D. Harkins and P. Kampanakis, "Mixing Preshared Keys in IKEv2 for Post-quantum Security." RFC-8784, May 2020, DOI 10.17487/RFC8784

[5] M. A. Nielsen and I. L. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press, 10th anniversary ed., 2010, ISBN: 978-1-107-00217-3

[6] L. K. Grover, "A fast quantum mechanical algorithm for database search", 28th Annual ACM Symposium on Theory of Computing, Philadelphia (PA, USA), May 22-24, 1996, pp. 212–219, DOI 10.1145/237814.237866

[7] G. Brassard, P. Høyer, M. Mosca, and A. Tapp, "Quantum Amplitude Amplification and Estimation", Quantum Computation and Information (S. J. J. Lomonaco and H. E. Brandt, eds.), vol. 305 of *Contemporary Mathematics*, pp. 53–74, American Mathematical Society (AMS), 2002. http://www.ams.org/books/conm/305/

[8] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography Standardization Call for Proposals." NIST Call for Proposals, December 2016, https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf

[9] D. J. Bernstein and T. Lange, "Post-quantum cryptography", Nature, vol. 549, no. 7671, 2017, pp. 188–194, DOI 10.1038/nature23461

[10] G. Alagic, D. C. Apon, D. A. Cooper, Q. H. Dang, T. T. Dang, J. M. Kelsey, Y.-K. Liu, C. A. Miller, D. Moody, R. Peralta, R. A. Perlner, A. J. Regenscheid, D. S. Robinson, and D. Smith-Tone, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process." NIST Internal Report 8413, July 2022, DOI 10.6028/NIST.IR.8413

[11] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard." FIPS 203, August 2024

[12] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard." FIPS 204, August 2024

[13] National Institute of Standards and Technology, "Stateless Hash-Based Digital Signature Standard." FIPS 205, August 2024

[14] O. Regev, "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography", Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, 2005, pp. 84–93, DOI 10.1145/1060590.1060603

[15] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Joux, M. Lachartre, S. Puchinger, and G. ZÃ©mor, "HQC: Hamming Quasi-Cyclic", Post-Quantum Cryptography – 10th Int. Conference (PQCrypto 2019), 2019, pp. 3–23, DOI 10.1007/978-3-030-25510-7_1

[16] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, A. Joux, A. Keren, C. A. Melchor, R. Misoczki, E. Persichetti,

S. Puchinger, J.-P. Tillich, and G. ZÃ©mor, "BIKE: Bit Flipping Key Encapsulation", Post-Quantum Cryptography – 10th Int. Conference (PQCrypto 2019), 2019, pp. 24–45, DOI 10.1007/978-3-030-25510-7_2

[17] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "The Internet Key Exchange Protocol Version 2 (IKEv2)." RFC-7296, October 2014, DOI 10.17487/RFC7296

[18] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC-5280, May 2008, DOI 10.17487/RFC5280

[19] S. Kent and R. Housley, "Composite Signatures for Use in Internet PKI." RFC-9510, February 2024, DOI 10.17487/RFC9510

[20] W. T. Polk, D. F. Dodson, J. M. O'Rourke, M. P. Souppaya, and W. C. Turner, "Guideline for Using Cryptographic Standards." NIST Special Publication 800-175B Revision 1, May 2020, DOI 10.6028/NIST.SP.800-175Br1

[21] W. Castryck and T. Decru, "An efficient key recovery attack on SIDH (preliminary version)", Eurocrypt 2023, Lyon (France), April 23-27, 2023, pp. 423–453, DOI 10.1007/978-3-031-30589-4_15

[22] J.-P. D'Anvers, L. Batina, S. Bhasin, F. Vercauteren, I. Verbauwhede, and A. Jati, "Cache-timing attacks on the modular lattice-based KEM Kyber", Cryptographic Hardware and Embedded Systems – CHES 2019, 2019, pp. 416–435, DOI 10.1007/978-3-030-25204-5_22

[23] R. Prates, G. F. T. Z'aba, F. T. de Mello, J.-P. D'Anvers, L. Batina, and E. Öztürk, "Dude, is my code constant-time?: a cache-timing analysis of ML-DSA (Dilithium)", 2022 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2022, pp. 386–394, DOI 10.1109/SBAC-PAD55450.2022.00062

[24] European Telecommunications Standards Institute (ETSI), "Migration strategies and recommendations to quantum-safe schemes." ETSI White Paper No. 38, June 2020, https://www.etsi.org/images/files/ETSIWhitePapers/ETSI_WP38_PQC_Migration_Strategies.pdf

[25] S. Kent and K. Seo, "Security Architecture for the Internet Protocol." RFC-4301, December 2005, DOI 10.17487/RFC4301

[26] S. Kent, "IP Authentication Header." RFC-4302, December 2005, DOI 10.17487/RFC4302

[27] S. Kent, "IP Encapsulating Security Payload (ESP)." RFC-4303, December 2005, DOI 10.17487/RFC4303

[28] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)." RFC-2409, November 1998, DOI 10.17487/RFC2409

[29] T. Kivinen and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)." RFC-3526, May 2003, DOI 10.17487/RFC3526

[30] S. Fluhrer, P. Kampanakis, and N. M, "Post-Quantum Authentication and Key Exchange for IKEv2." RFC-9242, May 2022, DOI 10.17487/RFC9242

[31] B. Berger, Y. Nir, P. Schwabe, and C. J. Tjhai, "IKEv2 Intermediate Exchange." RFC-9243, July 2022, DOI 10.17487/RFC9243

[32] C. J. Tjhai, "Using Post-Quantum Cryptography (PQC) in Encapsulating Security Payload (ESP)." RFC-9370, March 2023, DOI 10.17487/RFC9370

[33] S. Fluhrer, P. Kampanakis, and N. M, "Post-Quantum Authentication and Key Exchange for IKEv2." RFC-9242, May 2022, DOI 10.17487/RFC9242

[34] B. Berger, Y. Nir, P. Schwabe, and C. J. Tjhai, "IKEv2 Intermediate Exchange." RFC-9243, July 2022, DOI 10.17487/RFC9243

[35] C. J. Tjhai, "Using Post-Quantum Cryptography (PQC) in Encapsulating Security Payload (ESP)." RFC-9370, March 2023, DOI 10.17487/RFC9370

[36] R. Burkholder, "IPsec for Linux - strongSwan vs Openswan vs Libreswan vs other?." Server Fault, August 2016, https://serverfault.com/a/798638

[37] The Libreswan Project, "HOWTO: Using NSS with libreswan", 2024, https://libreswan.org/wiki/HOWTO:_Using_NSS_with_libreswan

[38] Oracle Corporation, "FIPS 140-2 Non-Proprietary Security Policy: Oracle Linux 7 Libreswan Cryptographic Module." NIST CMVP Security Policy Document, 2020, https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3699.pdf

[39] The strongSwan Project, "Plugin List - strongSwan Documentation", 2024, https://docs.strongswan.org/docs/latest/plugins/plugins.html

[40] U. Safdar, "Post Quantum Secure Strongswan with Liboqs." Medium, November 2023, https://medium.com/@umairsafdar768/post-quantum-secure-strongswan-with-liboqs-9659141ffbf9

[41] The strongSwan Project, "strongSwan - The OpenSource IPsec-based VPN Solution." https://www.strongswan.org/

[42] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "The Internet Key Exchange Protocol Version 2 (IKEv2)." RFC-7296, October 2014, DOI 10.17487/RFC7296

[43] The strongSwan Project, "strongSwan Architecture Overview." https://wiki.strongswan.org/projects/strongswan/wiki/ArchitectureOverview

[44] Open Quantum Safe, "Open Quantum Safe project website." Website, 2025, https://openquantumsafe.org/

[45] Open Quantum Safe, "liboqs: C library for quantum-safe cryptography." GitHub Repository, 2025, https://github.com/open-quantum-safe/liboqs

[46] Open Quantum Safe, "OQS forks: OpenSSL, strongSwan, and others." GitHub Organization, 2025, https://github.com/open-quantum-safe/

[47] R. Yavatkar, D. Pendarakis, and R. Guerin, "A Framework for Policy-based Admission Control." RFC-2753, January 2000, DOI 10.17487/RFC2753

[48] A. Lioy, "Electronic identity: delegated and federated authentication, policy-based access control." Lecture Slides, Politecnico di Torino, 2023, Internal course material.

[49] Open Policy Agent Project, "Open Policy Agent (OPA)." Website, 2023, https://www.openpolicyagent.org/

[50] T. Scott and T. Fisher, "Securing Kubernetes with OPA (Open Policy Agent)." Presentation at KubeCon + CloudNativeCon, November 2019

[51] C. Paquin, D. Stebila, and G. Tamvada, "Benchmarking Post-Quantum Cryptography in TLS and IKEv2", Post-Quantum Cryptography – 10th Int. Conference (PQCrypto 2019), 2019, pp. 250–272, DOI 10.1007/978-3-030-25510-7_14

[52] P. Kampanakis, D. Harkins, and S. Fluhrer, "Post-quantum IKEv2: An Experimental Evaluation", 2018 Network and Distributed System Security Symposium (NDSS), 2018, pp. 1–13, DOI 10.14722/ndss.2018.23197

[53] M. Wiggers, J.-F. Grote, C. Kühn, S. Ziegldrum, O. García-Morchón, and T. Güneysu, "Post-Quantum Cryptography in Practice: A TLS and IKEv2 Performance Analysis", 2020 IFIP Networking Conference (Networking), 2020, pp. 577–585, DOI 10.23919/Networking49061.2020.9141756

Appendix A

User Manual

A.1 System setup

This appendix describes how to install, configure, and operate the prototype developed for this thesis. The system is composed of a set of containerised services that collectively implement a policy-driven secure network based on `strongSwan` for IPsec tunnelling and the OPA as PDP. The deployment includes the policy enforcement gateway (`pep-gateway`) running `strongSwan` with the `ext-auth` plugin, one or more initiator hosts (e.g. `host-legacy`), application servers such as `banka`, auxiliary peers, and a suite of observability components including Prometheus, Grafana, and a dedicated decision-logger service.

A.1.1 Software prerequisites

Docker engine and Docker compose

Install Docker using the official repository. The following commands configure the repository, import Docker's signing key, and install both `docker-ce` and the Docker Compose plugin:

```
sudo apt update
sudo apt install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
    -o /etc/apt/keyrings/docker.asc
sudo install -m 0644 /etc/apt/keyrings/docker.asc \
    /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/etc/apt/keyrings/docker.asc] \
https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo \"$VERSION_CODENAME\") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io \
    docker-buildx-plugin docker-compose-plugin
```

Running Docker without `sudo`

To execute Docker as an unprivileged user, add your account to the `docker` group:

```
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Verify the installation:

```
docker run hello-world
```

Networking tools

Tools such as `tcpdump` and `tshark` are required for packet-level debugging and for later performance evaluation:

```
sudo apt update
sudo apt install tcpdump tshark
```

Project files

Clone the project repository:

```
git clone https://github.com/PQ-IPsec-Gateway/thesis.git <DIR>
cd <DIR>
```

The repository includes a `docker-compose.yml` file defining all components: the decision-logger, CTI service, OPA , Prometheus, Grafana, the policy-enforcing gateway, the various hosts, and auxiliary peers. Each service contains its own configuration directory, which provides the `strongSwan` and `swanctl` profiles for that container, among with certificates and daemon's scripts.

A.2 Building and starting the system

A.2.1 Building Docker images

From the project root directory, build all service images:

```
docker compose build
```

This command reconstructs all Docker images according to the definitions in `docker-compose.yml`. Rebuilding is required whenever Dockerfiles or build contexts change.

A.2.2 Launching the environment

Start all services in detached mode:

```
docker compose up -d
```

List running services:

```
docker compose ps
```

Stop or tear down the environment:

```
docker compose stop
docker compose down
```

A.2.3 Inspecting container status

To confirm correct initialisation:

```
docker compose logs <service>
```

Check that all essential components are active.

A.3 Loading strongSwan configuration

Each `strongSwan`-based component uses `swanctl` to load IKE/CHILD SA configurations and credentials via the VICI interface.

A.3.1 Loading configurations and credentials

Run the following inside each `strongSwan` container:

```
docker exec <container> swanctl --load-all
```

A.3.2 Listing available connection profiles

```
docker exec <container> swanctl --list-conns
```

This command outputs all configured IKE and CHILD connection definitions, such as `legacy-to-gateway`.

A.4 Establishing VPN tunnels

A.4.1 Initiating the IKE SA

As an outbound example, from the `host-legacy` container, initiate the IKE negotiation:

```
docker exec host-legacy swanctl --initiate --ike legacy-to-gateway
```

A.4.2 Inspecting Security Associations

Check established IKE and CHILD SAs:

```
docker exec host-legacy swanctl --list-sas
docker exec pep-gateway swanctl --list-sas
docker exec banka swanctl --list-sas
```

Entries should appear as `ESTABLISHED` with valid SPI pairs.

A.4.3 Testing connectivity

Verify the encrypted path:

```
docker exec host-legacy ping -c 4 10.100.0.10
```

ESP traffic can be inspected using `tcpdump` or `tshark`.

A.5 Inspecting logs and OPA decisions

A.5.1 Gateway authorisation logs

The gateway's `ext-auth` module delegates authorisation to OPA. View authorisation events:

```
docker exec pep-gateway tail -f /var/log/ext-auth.log
```

A.5.2 Child SA installation logs

`strongSwan` logs VICI-triggered CHILD SA installations:

```
docker exec pep-gateway \
  grep "VICI initiate" /var/log/vici-child-manager.log
```

A.5.3 OPA audit logs

The decision-logger stores all PDP decisions. Logs are available under:

```
docker exec decision-logger \
    tail -f /app/logs/decisions-YYYY-MM-DD.log
```

A.6 Visualising metrics with Grafana

Grafana is provided for real-time observability. After deployment, access: `http:// localhost:3000`. The default credentials are `admin/pqgw_admin`. Import dashboards via:

Dashboards → *Import* → *Upload JSON*.

Dashboards display `strongSwan` metrics, OPA decision statistics and network traffic counters.

A.7 Testing and troubleshooting

A.7.1 Performance Tests

The performance experiments described in this thesis are each associated with a dedicated branch of the project repository, namely `test-classic`, `test-ike-fragmentation` and `test-benchmark`. Each branch contains a minimally adapted variant of the system, together with lightweight instrumentation and helper scripts, allowing the corresponding test campaigns to be executed in a controlled and reproducible manner.

`test-classic`

The experiment is launched with:

```
bash scripts/simple-ike-comparison.sh
```

Internally, the script deploys the appropriate `swanctl` configuration, restarts the relevant containers, and reloads the `strongSwan` settings before triggering the handshake. A packet capture is started on the gateway, and the resulting PCAP is processed with a small Python script that extracts packet sizes, IKE message types, and IP fragmentation. The outputs from the normal and childless runs are then compared to quantify time-level and byte-level differences.

`test-ike-fragmentation`

This branch provides a tailored environment for analysing fragmentation behaviour and per-phase timing during the IKE exchange. The suite is executed through:

```
./scripts/capture-and-analyze-ike.sh pep-gateway gw-wan-out-bankA 198.51.100.2
```

The script clears existing SAs, initiates a timestamped `tshark` capture on ports 500/4500, triggers the handshake, and subsequently parses the PCAP to recover message boundaries, intermediate ADDKE exchanges, and precise request/response timings. The output provides fine-grained measurements of per- phase latency and fragmentation, enabling comparisons between multi-KEM and single-KEM configurations.

`test-benchmark`

The most comprehensive evaluation is conducted in the `test-benchmark` branch, which correlates three independent data sources: raw PCAP captures, `strongSwan` VICI logs, and OPA audit records. The benchmark is executed using:

```
./scripts/run-benchmark.sh
```

The script resets the environment, and based on the level specified in the code, initiates a full tunnel establishment, captures all on-wire traffic, and records state transitions via VICI while OPA produces detailed timing reports for IKE and CHILD policy decisions. These heterogeneous data streams are then aligned to derive end-to-end timing, traffic volume, fragmentation counts, and policy-evaluation metrics, all of which are stored in structured JSON format for subsequent analysis.

A.7.2 Rebuilding and cleaning the environment

Rebuild images:

```
docker compose build --no-cache
```

Remove containers and volumes:

```
docker compose down --volumes
docker compose up -d
```

A.7.3 Diagnosing connection issues

- ensure each `strongSwan` container has reloaded its configuration via `swanctl --load-all`;
- inspect `/var/log/charon.log` for negotiation errors;
- verify that OPA policies authorise the requested connection;
- confirm certificate trust chains and identities match on both peers.

Appendix B

Developer's Reference Guide

This appendix describes the internal organisation of the thesis prototype from a developer's standpoint. It concentrates on the integration between `strongSwan` and `Open Policy Agent` in the `pep-gateway`, the custom patches applied to `strongSwan`, and the policy model that governs the creation and adaptation of tunnels.

The aim of this chapter is to equip a future maintainer with sufficient detail to understand where and how authorisation decisions are enforced, extend the deployment with additional legacy hosts or external peers, adjust the security levels and cryptographic algorithms enforced by the gateway and keep the local patches aligned with upstream `strongSwan`.

B.1 strongSwan–OPA integration

The policy enforcement point is realised in the `pep-gateway` container. From the perspective of `strongSwan`, the integration is built on three main elements:

1. a patched instance of the `ext_auth` plugin in `strongSwan`, implemented in `src/libcharon/plugins/ext_auth/ext_auth_listener.c`;
2. an helper, `opa-check-auth`, which bridges IKEv2's execution environment with the PDP;
3. auxiliary scripts and services inside `pep-gateway` (e.g. `vici-child-manager` and the `updown` hooks) that implement tunnel provisioning and re-provisioning according to OPA's decisions.

B.2 strongSwan `ext_auth` patch

B.2.1 File location and high-level responsibilities

The core of the C-level integration resides in the modified `ext_auth_listener` implementation:

- file: `src/libcharon/plugins/ext_auth/ext_auth_listener.c`;
- plugin: `ext_auth`, loaded by the IKE daemon `charon`.

The initial part of the patch extends the include set and introduces a small in-memory structure used to record pending OPA hints, indexed by the IKE `unique_id`:

Listing B.1. Additional includes and hint entry definition in `ext_auth_listener.c`

```
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <ctype.h>

#include <utils/chunk.h>
#include <utils/enum.h>
#include <collections/linked_list.h>
#include <threading/mutex.h>
#include <encoding/message.h>
#include <crypto/proposal/proposal.h>
#include <crypto/transform.h>
#include <crypto/prfs/prf.h>
#include <crypto/key_exchange.h>
#include <credentials/certificates/certificate.h>
#include <credentials/auth_cfg.h>

typedef struct {
    uint32_t unique_id;
    char *required_ke;
    bool pending_notify;
} hint_entry_t;

#define NOTIFY_REQUIRED_LEVEL 0xA001

typedef struct private_ext_auth_listener_t private_ext_auth_listener_t;

struct private_ext_auth_listener_t {
    ext_auth_listener_t public;

    /** Path to authorization program */
    char *script;

    /** Pending hints (hint_entry_t*) keyed by ike unique_id */
    linked_list_t *hints;

    /** Protect access to hints */
    mutex_t *mutex;
};
```

The `hints` list keeps, for each IKE_SA (identified by `unique_id`), the pending *required level* value received from OPA and a boolean flag `pending_notify` which is later consulted when deciding whether to attach a custom `NOTIFY` payload to the outbound IKE_AUTH.

Regarding the peer certificate extraction, the helper function `acquire_peer_cert()` iterates over the current `auth_cfg` chain to retrieve the peer's end-entity certificate, which is then exported both inline (as PEM in an environment variable) and via a temporary file:

Listing B.2. Peer certificate acquisition from the current auth round

```
static certificate_t* acquire_peer_cert(ike_sa_t *ike_sa)
{
    enumerator_t *enumerator;
    auth_cfg_t *cfg;
    certificate_t *cert = NULL;

    enumerator = ike_sa->create_auth_cfg_enumerator(ike_sa, FALSE);
```

```
    if (!enumerator)
    {
        return NULL;
    }
    while (enumerator->enumerate(enumerator, &cfg))
    {
        cert = cfg->get(cfg, AUTH_RULE_SUBJECT_CERT);
        if (cert)
        {
            break;
        }
    }
    enumerator->destroy(enumerator);
    return cert;
}
```

Later in the authorisation path, this certificate is encoded as PEM and exposed to the external script:

Listing B.3. Exporting the peer certificate to the environment and a temporary file

```
peer_cert = acquire_peer_cert(ike_sa);
if (peer_cert && peer_cert->get_encoding(peer_cert, CERT_PEM, &pem))
{
    push_env(envp, countof(envp), "PLUTO_PEER_CERT_PEM=%.*s",
             (int)pem.len, pem.ptr);
    char template[] = "/tmp/ext-auth-peer-XXXXXX.pem";
    int fd = mkstemp(template);
    if (fd >= 0)
    {
        ssize_t written = write(fd, pem.ptr, pem.len);
        close(fd);
        if (written == (ssize_t)pem.len)
        {
            push_env(envp, countof(envp), "PLUTO_PEER_CERT=%s", template);
            tmp_cert_path = strdup(template);
        }
        else
        {
            unlink(template);
        }
    }
    chunk_free(&pem);
}
```

The variables `PLUTO_PEER_CERT_PEM` and `PLUTO_PEER_CERT` are then consumed by `opa-check-auth`, which embed the certificate into the JSON input passed to the PDP.

Moreover, the patch also enriches the authorisation environment with detailed information on the negotiated algorithms. This information is derived from the `proposal_t` associated with the IKE_SA:

Listing B.4. Exporting negotiated PRF and key exchange methods

```
proposal = ike_sa->get_proposal(ike_sa);
if (proposal)
{
    u_int idx;
```

```
    uint16_t alg = 0, key_size = 0;
    const char *name;

    if (proposal->get_algorithm(proposal, PSEUDO_RANDOM_FUNCTION,
                                &alg, &key_size))
    {
        (void)key_size;
        name = enum_to_name(pseudo_random_function_names, alg);
        if (name && *name)
        {
            push_env(envp, countof(envp), "PLUTO_IKE_PRF=%s", name);
        }
    }
    if (proposal->get_algorithm(proposal, KEY_EXCHANGE_METHOD,
                                &alg, &key_size))
    {
        (void)key_size;
        name = enum_to_name(key_exchange_method_names, alg);
        if (name && *name)
        {
            push_env(envp, countof(envp), "PLUTO_IKE_DH=%s", name);
        }
    }
    for (idx = 0; idx < 3; idx++)
    {
        transform_type_t type = ADDITIONAL_KEY_EXCHANGE_1 + idx;
        if (proposal->get_algorithm(proposal, type, &alg, &key_size))
        {
            (void)key_size;
            name = enum_to_name(key_exchange_method_names, alg);
            if (name && *name)
            {
                push_env(envp, countof(envp),
                         "PLUTO_IKE_KE%d=%s", idx + 1, name);
            }
        }
    }
}
```

Consequently, each authorisation decision receives an environment containing:

- IKE identities (`IKE_LOCAL_ID`, `IKE_REMOTE_ID`);
- the negotiated PRF (`PLUTO_IKE_PRF`);
- the negotiated key-exchange / DH group (`PLUTO_IKE_DH`);
- up to three additional key-exchange identifiers (`PLUTO_IKE_KE1`, `PLUTO_IKE_KE2`, `PLUTO_IKE_KE3`);
- the peer certificate in PEM form.

Informations used by OPA to compute the security level, as described in the policy modules.

B.2.2 OPA hints and `REQUIRED_LEVEL` notify

The exchange between OPA and the gateway is intentionally richer than a simple boolean *allow/-deny* outcome. The helper script may also emit an "OPA hint" indicating the *required level* for

a given IKE_SA. The patched `ext_auth` listener parses this hint and stores it in a `hint_entry_t` structure:

Listing B.5. Parsing OPA hints from the helper script

```
char *hint = strstr(resp, "OPA_HINT");
if (hint)
{
    uint32_t hint_id = 0;
    hint_level[0] = '\0';
    if (sscanf(hint, "OPA_HINT unique_id=%u required_ke=%31s",
              &hint_id, hint_level) >= 2)
    {
        DBG1(DBG_CHD, "ext-auth: parsed OPA hint unique_id=%u (local=%u) level
             '%s'",
             hint_id, unique_id, hint_level);
        if (hint_id == unique_id)
        {
            DBG1(DBG_CHD, "ext-auth: parsed hint for IKE_SA %u (hint=%u) level
                '%s'",
                 unique_id, hint_id, hint_level);
            store_required_ke(this, hint_id, hint_level);
        }
    }
}
```

If the script returns a failure code (i.e. the policy engine rejects the current level), the listener marks the hint as pending and defers the actual notification to the *next* outbound IKE_AUTH message:

Listing B.6. Scheduling and attaching the REQUIRED_LEVEL notify

```
if (*success)
{
    clear_hint(this, unique_id);
}
else
{
    set_pending_notify(this, unique_id, TRUE);
}

/* ... */

METHOD(listener_t, message, bool,
    private_ext_auth_listener_t *this, ike_sa_t *ike_sa, message_t *message,
    bool incoming, bool plain)
{
    uint32_t unique_id = ike_sa->get_unique_id(ike_sa);

    if (!plain)
    {
        return TRUE;
    }

    if (incoming && message->get_exchange_type(message) == IKE_AUTH)
    {
        /* incoming REQUIRED_LEVEL notify: store the required level */
        notify_payload_t *notify = message->get_notify(message,
```

```
                                                    NOTIFY_REQUIRED_LEVEL);
        /* parse value and call store_required_ke(...) */
        /* ... */
        return TRUE;
    }

    if (!incoming && message->get_exchange_type(message) == IKE_AUTH)
    {
        hint_entry_t *entry;

        this->mutex->lock(this->mutex);
        entry = find_hint(this, unique_id);
        if (entry && entry->pending_notify && entry->required_ke &&
            entry->required_ke[0])
        {
            chunk_t payload = chunk_create(entry->required_ke,
                                           strlen(entry->required_ke));
            DBG1(DBG_CHD, "ext-auth: attaching required level '%s' for IKE_SA %
                u",
                 entry->required_ke, unique_id);
            message->add_notify(message, FALSE,
                                NOTIFY_REQUIRED_LEVEL, payload);
            this->hints->remove(this->hints, entry, NULL);
            this->mutex->unlock(this->mutex);
            destroy_hint_entry(entry);
            return TRUE;
        }
        this->mutex->unlock(this->mutex);
    }
    return TRUE;
}
```

The constructor for the `ext_auth` listener is extended to allocate and initialise both the hint list and its mutex:

Listing B.7. Listener creation with hint list initialization

```
METHOD(ext_auth_listener_t, create, ext_auth_listener_t*,
    char *script)
{
    private_ext_auth_listener_t *this;

    INIT(this,
        .public = {
            .listener = {
                .authorize = _authorize,
                .message = _message,
            },
            .destroy = _destroy,
        },
        .script = script,
        .hints = linked_list_create(),
        .mutex = mutex_create(MUTEX_TYPE_DEFAULT),
    );

    if (!this->hints || !this->mutex)
    {
```

```
        if (this->hints)
        {
            this->hints->destroy(this->hints);
        }
        if (this->mutex)
        {
            this->mutex->destroy(this->mutex);
        }
        free(this);
        return NULL;
    }

    return &this->public;
}
```

The corresponding `destroy` method is responsible for releasing any remaining hint entries and deallocating the synchronisation primitives.

B.3 `opa-check-auth` and helper scripts

B.3.1 Authorisation script interface

The `opa-check-auth` script is called by `ext_auth` with the environment described above. From a developer's viewpoint, the essential contract is:

- exit code `0` ⇒ authorisation *granted*;
- non-zero exit code ⇒ authorisation *denied*;
- optional diagnostic and hint lines printed on `stdout`, for example:

Listing B.8. Example output from `opa-check-auth`

```
OPA_DECISION allow=true level="L2"
OPA_HINT unique_id=42 required_ke=L3 required_sig=L2
```

The `OPA_HINT` line is parsed by the `ext_auth` patch as shown in Listing B.5, eventually leading to a `REQUIRED_LEVEL` notify being attached to an outbound IKE_AUTH.

B.4 Connection naming and tunnel provisioning

B.4.1 Naming conventions for connections and hosts

To simplify routing and provisioning logic, connection names in the `strongSwan` configuration are chosen with specific semantics rather than arbitrarily. The project adopts the following conventions:

- connections between the legacy host and the gateway explicitly contain the string `"legacy"` in their name;
- connections between the gateway and external peers explicitly encode the direction using suffixes such as `"outbound"` or `"inbound"`.

These conventions are consumed by:

- the `tunnel-provisioning` logic, which use this information, and also the knowledge of the traffic selectors, to understand that, there is an opening for an outbound connection from the legacy subnet and provide a "just-in-time" post-quantum tunnel counterpart;
- the policy layer, which use this information as additional helper to map connection names and peer identifiers to service classes and security levels.

By embedding semantics directly in connection names (legacy vs. peer, inbound vs. outbound), the provisioning logic avoids relying solely on IP addressing to infer the role and direction of each tunnel.

B.5 OPA policy modules

The core IKE establishment policy logic is expressed in four Rego modules under the `ike` namespace:

- `ike.service_classes` (`service_classes.rego`);
- `ike.establishment` (`ike_establishment.rego`);
- `ike.child_templates` (`child_templates.rego`);
- `ike.certificates` (`certificate_validation.rego`).

Collectively, these modules:

1. classify internal subnets and external partners into *service classes*;
2. define the required IKE key-exchange level (`KE-Lx`) for each internal-external pair;
3. choose the appropriate CHILD_SA template;
4. validate the peer certificate according to post-quantum signature and public-key requirements.

B.5.1 `service_classes.rego`

The module `ike.service_classes` defines a declarative mapping from IP prefixes and peers to service classes and cryptographic requirements:

Listing B.9. Top-level mapping in `service_classes.rego`

```
package ike.service_classes

import rego.v1

# INTERNAL SERVICE CLASSES - Subnet-based Classification

internal_service_classes := {
    "legacy_internal": {
        "subnet": "10.200.0.0/24",
        "security_profile": "legacy",
        "crypto_capabilities": ["RSA-2048", "ECDH-P256", "ECDSA-P256"],
        "description": "Internal legacy hosts using traditional cryptography",
    },
```

```
    ...
}

external_partner_classes := {
    "bankA": {
        "subnets": ["198.51.100.10/32"],
        "service_type": "payments",
        "min_ke_level": "KE-L3",
        "min_sig_level": "SIG-L3",
        "min_pubkey_level": "SIG-L3",
        "allowed_issuers": {
            "C=CH, O=Cyber, CN=Cyber Root CA",
            "O=PQ-Gateway Lab, CN=PQ-Gateway IKE CA PQ",
            ...
        },
        "description": "External bank A (payments, high assurance)"
    },
    ...
}
```

Rather than enumerating each internal host individually, internal classes are defined per *subnet* (for example `10.200.0.0/24` for legacy systems). Each external partner (e.g. `bankA`, `partnerB`, `opsC`) specifies:

- one or more IP prefixes;
- a `service_type` (for example *payments*, *erp*, *hr*);
- minimum levels `min_ke_level`, `min_sig_level`, `min_pubkey_level`;
- a set of `allowed_issuers` (distinguished names of trusted CAs).

B.5.2 `ike_establishment.rego`

The `ike.establishment` module is the main decision point. It imports the other rego modules, to enrich the decision, maintain modularity and fast changes.

Listing B.10. Imports in `ike_establishment.rego`

```
package ike.establishment

import rego.v1
import data.ike.certificates
import data.ike.child_templates
import data.ike.peer_mapping
import data.ike.service_classes
```

Mapping IKE suites to KE levels For each IKE_SA, a canonical string representing the full suite is constructed and used as a key into the `ke_suites` mapping:

Listing B.11. KE suite mapping in `ike_establishment.rego`

```
# Each KE level is defined by EXACT suite combinations, not individual
    algorithms

ke_suites := {
```

```
    # KE-L1 Suites (NIST Level 1 - HR, low-sensitivity)
    "aes128gcm16-sha256-ecp256-mlkem512-none-none": "KE-L1",
    "aes128gcm16-sha256-modp2048-mlkem512-none-none": "KE-L1",

    # KE-L2 Suites (NIST Level 3 - ERP, moderate)
    "aes192gcm16-sha384-ecp256-mlkem768-none-none": "KE-L2",
    "aes192gcm16-sha384-ecp384-mlkem768-none-none": "KE-L2",
    "aes192gcm16-sha384-x25519-mlkem768-none-none": "KE-L2",

    # KE-L3 Suites (NIST Level 3+ - Payments, enhanced classic)
    "aes256gcm16-sha512-ecp384-mlkem768-none-none": "KE-L3",
    "aes256gcm16-sha512-ecp521-mlkem768-none-none": "KE-L3",
    ...
}
```

A helper, `build_suite_string(input.ike)`, concatenates the negotiated algorithms into the string used to query `ke_suites`. The resulting value is the *achieved* level, `ke_achieved`.

Partner-specific templates For each partner and level, fully specified templates are provided.

Listing B.12. Example inbound template for `partnerB`

```
responder_templates := {
    "partnerB": {
        "KE-L2": {
            "name": "inbound-legacy-erp-L2",
            "local_ts": "10.200.0.0/24",
            "remote_ts": "198.51.100.11/32",
            "child_level": "CHILD-L2",
            "esp_proposals": child_security_levels["CHILD-L2"].esp_proposals,
            "rekey_time": "90s",
            "reqid": 103,
            "start_action": "none",
            "updown": "/usr/local/sbin/updown-verifier.sh",
            "description": "PartnerB inbound ERP tunnel (L2 security)",
        },
        "KE-L3": {
            "name": "inbound-legacy-erp-L3",
            ...
        },
    },
    ...
}
```

This architecture permits the introduction of additional partners or child levels without requiring alterations to the overall template configuration, hence enabling modifications to minor components without compromising the integrity of the whole system.

B.5.3 `certificate_validation.rego`

The `ike.certificates` module encodes the post-quantum requirements for X.509 certificates used during IKE exchanges.

The module defines mappings from OIDs to signature and public-key algorithms and their base levels:

Listing B.13. Example OID → algorithm/level mapping

```
sig_alg_base_level := {
    # Pure ML-DSA (NIST standardized)
    "mldsa44": "SIG-L1",
    "mldsa65": "SIG-L2",
    "mldsa87": "SIG-L3",

    # ML-DSA44 composites (NIST Level 2 - HR, low-sensitivity)
    "mldsa44-rsa2048-pss-sha256": "SIG-L1",
    "mldsa44-rsa2048-pkcs15-sha256": "SIG-L1",
    "mldsa44-ecdsa-p256-sha256": "SIG-L1",
    "mldsa44-ed25519-sha512": "SIG-L1-SUF",
    ...
}

sig_level_hierarchy := {
    "SIG-L0": 0,
    "SIG-L1": 1,
    "SIG-L1-SUF": 2,
    "SIG-L2": 3,
    "SIG-L3": 4,
    "SIG-L3-SUF": 5,
}
```

The functions `get_sig_alg(oid)` and `get_pubkey_alg(oid)` map the OIDs present in the certificate to canonical algorithm names.

Per-peer requirements (minimum signature and public-key levels) are derived from the previously described entries in `service_classes.external_partner_classes`. The module exposes, among others, the following helper:

Listing B.14. Peer requirements in `certificate_validation.rego`

```
get_cert_requirements(peer_name) := reqs if {
    partner := service_classes.external_partner_classes[peer_name]
    reqs := {
        "min_sig_level": partner.min_sig_level,
        "min_pubkey_level": partner.min_pubkey_level,
        "allowed_issuers": partner.allowed_issuers,
    }
}
```

Evolving the security model To adjust the *ordering of levels* or the algorithms associated with each level, it is sufficient to update:

- `ke_suites` and the KE level associations in `ike_establishment.rego`;
- `child_security_levels` and `ke_to_child_level` in `child_templates.rego`;
- `sig_alg_base_level`, `pubkey_base_level` and `sig_level_hierarchy` in `certificate_validation.rego`.

The Rego logic that combines these levels (consistency between IKE, CHILD and certificate requirements) remains unchanged. Only the configuration tables are modified, keeping the system extensible and relatively easy to maintain.

The validation logic is deliberately stable: the semantics of "what counts as L1/L2/L3" are concentrated in the level lists and OID mappings. This makes it possible to refine the security

profile (e.g. by deprecating an algorithm or promoting a stronger suite) without touching the strongSwan patch or the control-plane scripts.

B.6 Applying patches and extending the system

B.6.1 Patch directory in `pep-gateway`

The `pep-gateway` container includes an internal `patches` directory which holds all modifications applied on top of the upstream `strongSwan` sources.

During Docker image construction, these patches are applied to the `strongSwan` sources prior to compilation.

To introduce a new C-level modification (e.g. extending the notifier behaviour or adjusting the exported environment), the recommended workflow is:

1. locate the source file to be modified within the container build context;
2. copy it to a "`_mod`" variant, apply the desired edits, and generate a unified diff;
3. store the resulting patch under `pep-gateway/patches`;
4. rebuild the Docker image so that the patch is applied during the build.

A generic pattern is:

Listing B.15. Generic patch creation workflow

```
cd /tmp
git clone --depth 1 --branch 6.0.0beta6 \
  https://github.com/strongswan/strongswan.git
cd strongswan

cp src/libcharon/plugins/ext_auth/ext_auth_listener.c \
   src/libcharon/plugins/ext_auth/ext_auth_listener_mod.c

# Apply your modifications to ext_auth_listener_mod.c

diff -Naur \
  src/libcharon/plugins/ext_auth/ext_auth_listener.c \
  src/libcharon/plugins/ext_auth/ext_auth_listener_mod.c \
  > ext-auth-listener.patch

mv ext-auth-listener.patch /path/to/pep-gateway/patches/

cd /tmp && rm -rf strongswan

# rebuild the Docker image to apply the patch
cd /path/to/pep-gateway
docker build -t pep-gateway
```

B.6.2 Adding new connections and hosts

From a configuration perspective, adding a new legacy subnet or external peer involves:

1. **defining the connection in strongSwan**, be compliant to the naming conventions;
2. **updating** `service_classes.rego` to:
   - register the new legacy subnet or partner;
   - configure its default and permissible security levels;
   - optionally define per-subnet and per-partner requirements.

Insert something here ...