Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 13 August 2026

P.J. Nsiangani, Ed.
ACE Working Group
9 February 2026

ASL Authenticated Secure Layer Protocol
draft-nsiangani-authenticatedsecuredlayer-00

Abstract

Complete ASL draft embedded below as imported text. Replace this abstract with the structured version later if desired.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 August 2026.

Copyright Notice

Table of Contents

1. Imported Internet-Draft Content 2
2. References 63
Author's Address 63

1. Imported Internet-Draft Content

ACE Working Group Author: P.J. Nsiangani Ed.
Internet-Draft draft-nsiangani-authenticatedsecuredlayer-00
Intended status: Proposed Standard
Expires: September 2026 09 February 2026

ASL Authenticated Secure Layer Protocol

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This document is subject to the rights, licenses and restrictions contained in BCP 78 and BCP 79, and except as set forth therein, the authors retain all their rights.

This document is provided under the terms of the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

Copyright Notice

This Internet-Draft will expire in September 2026.

Abstract

This document specifies ASL (Authenticated Secure Layer), a cryptographic protocol for establishing an authenticated and encrypted communication session based on a novel arithmetic primitive called phi-ns-x (also known as ns-mod). ASL provides a complete handshake and data transport layer without reliance on traditional algorithms like RSA, Diffie-Hellman, AES or SHA. Instead, it builds on the Phi-Ns cryptosystem, which defines a structured quadratic relation between primes and achieves security through non-linear modular arithmetic and combinatorial complexity.

ASL includes an initial authentication and key exchange mechanism where a server public key (composed only of specific allowed elements such as a prime modulus and state parameters) is used by a client to establish shared secrets. The protocol uses a stateful encryption transform (phi-ns-x) for data, with each message round incorporating a public state value that changes unpredictably. Shared secret parameters control this state evolution, preventing man-in-the-middle and injection attacks by ensuring that only parties with knowledge of the secrets can produce valid ciphertexts.

Table of Contents

1. Introduction 3
2. Terminology 5
3. The Phi-ns-x Cryptographic Primitive 7
3.1. Fundamental Equations and Structure 7
3.2. Sequential Encryption Recurrence (ns-mod) 9
3.3. Optional Layer Injection and Masking 11
3.4. Cryptographic Properties of Phi-ns-x 12
4. ASL Protocol Specification 14

4.1. Roles and Key Components .....
14
4.2. Session Establishment (Handshake) .....
15
4.3. Round-Based Encrypted Messages .....
18
4.4. Message Formats and Encoding .....
20
5. Example Use Cases .....
22
5.1. HTTPS Without TLS (Web Security) .....
22
5.2. Blockchain Transactions .....
23
5.3. Implicit Client Identity Authentication .....
24
5.4. Secure Messaging .....
25
5.5. Encrypted Tunneling (VPN) .....
26
6.Security Considerations 27
6.1. Man-in-the-Middle (MITM) .....
27
6.2. Collision Resistance .....
28
6.3. Secret Leakage and Oracle-Free Design .....
28
6.4. Post-Quantum Resilience .....
29
6.5. Unpredictability of States .....
30
7. IANA Considerations .....
31
8. References .....
32
8.1. Normative References .....
32
8.2. Informative References .....
33

1. Introduction

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119] and RFC 8174 [RFC8174] when, and only when, they appear in all capitals, as shown here.

ASL (Authenticated Secure Layer) is a protocol that establishes an authenticated and encrypted channel between two parties (typically a client and a server) using only modular arithmetic operations for cryptographic security. ASL is designed as an alternative to TLS and similar protocols, particularly in contexts where post-quantum security, small key sizes, or avoidance of legacy primitives (RSA, ECC, AES, SHA) are desired. The protocol leverages the properties of the Phi-Ns cryptosystem [PhiNs] and its sequential encryption primitive (phi-ns-x) to achieve confidentiality and integrity in a novel way.

The foundation of ASL’s security is the Phi-Ns hardness assumption. Phi-Ns is an asymmetric cryptographic primitive based on the structured decomposition of a quadratic difference between two primes. In Phi-Ns, given two large primes p and q, the relation q^2 - p^2 = T defines a composite gap T (Eq0). This gap T is factored in a controlled way into a triple of secret values (a, b, R), where a and b are small exponents of 2 and 3 respectively, and R is a remaining large cofactor. The tuple (a, b, R) is then encoded in a randomized manner. An attacker who only knows one of the primes (the public key) and a secure commitment to the encoded structure cannot feasibly recover the other prime or the tuple, because there is no oracle or structural hint to guide the search. The problem reduces to blindly guessing the correct prime and internal factors among an astronomically large set of possibilities. The Phi-Ns mechanism does not rely on integer factorization or discrete logarithm; instead its security arises from the ambiguity and combinatorial explosion of reconstructing the hidden structure from minimal public information.

ASL uses this primitive in a session context. The protocol is composed of two phases: an **authentication and key exchange phase** (handshake), followed by a **secure data transport phase**. During the handshake, the server proves its identity and the two parties establish shared secret

parameters that will drive the encryption of messages. The handshake avoids the need for any external certificate signatures or third-party trust anchors beyond a one-time publication of the server’s public key (and an associated commitment value). The client authenticates the server by verifying a static masked value derived from q^2 (the servers hidden prime square), ensuring the server possesses the corresponding secret.
Once authenticated, the client and server use phi-ns-x to exchange or derive symmetric secrets such as trajectory steps for the encryption state. In the data transport phase, every message is encrypted using a **sequential modular transform**: each encryption round employs the phi-ns-x recurrence with evolving public state values. Only the legitimate parties (having the shared secrets from the handshake) can maintain and predict the correct state sequence, making it infeasible for an outsider to inject valid messages or decipher the traffic.

Unlike TLS which depends on PKI certificates, symmetric ciphers (AES), and hash-based HMAC for integrity, ASL achieves all these objectives through pure arithmetic. Authenticity is ensured by the possession of secret primes/factors and the correct derivation of state; confidentiality is provided by the non-linear encryption transform; and integrity is implicitly guaranteed since any modification to a ciphertext or state by an adversary results in a decryption failure (the decrypted data will not match the expected structure, and the receiver can detect this). The protocol is suitable for environments where conventional cryptography is hard to deploy: e.g., IoT devices with constrained resources (small keys and low processing overhead), or specialized scenarios like blockchain transactions where on-chain data must be encrypted yet verifiable.

The remainder of this document is organized as follows. Section 2 defines terminology used in the specification. Section 3 describes the

phi-ns-x cryptographic primitive in detail, including its mathematical definitions (Eq0 through Eq6) and cryptographic properties (non-linearity, entropy, direct non-factorizability). Section 4 specifies the ASL protocol: the roles of client and server, the format of public keys and messages, the handshake exchange (including the use of a q^2_masked value for static signature), the concept of a rotating state in encrypted messages, and the handling of shared secrets. In Section 5, we present several use cases to illustrate how ASL can be applied: secure web communication without TLS, blockchain with encrypted payloads, implicit client identity verification via key usage, end-to-end secure messaging, and encrypted tunneling. Section 6 discusses security considerations, including resistance to MITM, collision and leakage analysis, post-quantum aspects, and the unpredictability of protocol state. Finally, Section 7 notes IANA considerations (none) and Section 8 provides references to relevant literature and standards.

By introducing ASL as a formal Internet-Draft, our goal is to enable further scrutiny and implementation of this protocol. The design emphasizes a self-contained approach to security where all critical components are derived from a single hard problem (the phi-ns structured gap) and its associated modular arithmetic transform. This approach can complement ongoing efforts in post-quantum cryptography by providing a session-layer construction that is not only resistant to known quantum attacks but also efficient and compact.

2. Terminology

This section defines terms and notation used throughout this document. All definitions are normative for the ASL protocol and the underlying phi-ns cryptographic primitive.

Prime: A positive integer greater than 1 whose only divisors are 1 and itself.

p: A secret prime in the Phi-Ns cryptosystem. In the PK-q mode of operation
(public key = q), p is the private key. In the PK-p mode (public key = p),
p is public while q is kept private. p participates in the core equation q^2 - p^2 = T.

q: A prime associated with p in the Phi-Ns system. In PK-q mode, q is the public
key; in PK-p mode, q is private. q is chosen such that q^2 - p^2 = T for some composite T.

T: The quadratic gap defined by T = q^2 - p^2 (Eq0). T is a strictly positive composite
integer resulting from the difference of the two prime squares. By construction, T is
divisible by 4 (since q^2 - p^2 = (q - p)*(q + p), and q and p are either both odd or both even;
for primes >2, both are odd, so q-p and q+p are even, making T divisible by 4).

a: The exponent of 2 in the factorization of T. Specifically, a is the largest non-negative
integer such that 2^a divides T, but 2^(a+1) does not. The value ‘a‘ is part of the secret
tuple and is not revealed publicly.

b: The exponent of 3 in the factorization of T (after removing the 2^a factor). Specifically,
b is the largest non-negative integer such that 3^b divides T / 2^a, but 3^(b+1) does not. This
value is also secret (part of the tuple) and not revealed. (Note: In the context of the encryption
recurrence, the symbol b_i will be used to denote a different concept "door" values for injection
which is unrelated to this exponent b. The usage will be clear from context.)

R: The remaining composite factor of T after extracting 2^a * 3^b. That is, R = T / (2^a * 3^b).
R is an odd composite (not divisible by 2 or 3) and may consist of one or several prime factors.
R is secret.

abR: The structured decomposition of T into the tuple (a, b, R).

This triple encapsulates all
secret factors derived from the quadratic gap. Knowledge of abR (along with whichever prime was
kept private) is equivalent to knowing the full private key.

Serialized abR: A randomized encoding of the (a, b, R) tuple. In Phi-Ns key generation, the atoms
(prime factors) comprising R are first partially factored (e.g., small factors extracted), then all
resulting factors ("atoms") are permuted in a pseudorandom order and grouped into blocks. The final
encoded form includes the values a, b, a salt, the number of blocks, and the concatenated blocks of
atoms in permuted order. This serialized abR is committed (e.g., via a hash or a public checksum)
to serve as a validation of the key without revealing structure.

Recursive decomposition: An optional process in Phi-Ns where the secret prime p itself is
recursively defined by a smaller instance of the problem. For example, p^2 = p2^2 + T2, with its
own (a2, b2, R2) decomposition. Recursion can be applied multiple times, each time increasing the
entropy of the private key without increasing public key size. If recursion is used, the abR tuple
implicitly contains nested layers of factors. ASL and phi-ns-x operate at the top level; recursion
mainly affects key generation and security strength.

PK-q: A public key exposure mode in which q is the public key and p (along with abR) is kept
private. Attackers see q and a commitment to the serialized abR. They must attempt to guess the
correct p (and structure) that yields that commitment. This is the "standard" mode analogous to
most public-key systems where the public key is used for encryption.

PK-p: A public key exposure mode where p is public and q (with abR) is private. Attackers see p
and the commitment, but have no direct information about q or T. This mode offers higher security
because the attacker does not even know the magnitude or sign of T = q^2 - p^2; q could be slightly
larger than p or vastly larger, introducing more uncertainty. However, since encryption operations
usually require knowledge of q (as an anchor for computations), PK-p mode in ASL is supported by
providing alternate public parameters (such as q^2_masked and related

values) that allow encryption
without revealing q.

M: A large prime modulus under which phi-ns-x sequential encryption operations are performed.
All arithmetic for message encryption in ASL takes place in the finite field Z_M. M is chosen such
that it is large enough to accommodate message space and security requirements (often on the order
of 2^N for some N, or a safe prime). M is public and is one of the parameters included in the
server’s public key profile.

g: A generator of the multiplicative group Z_M^*. The value g is used to advance public states
in a cyclic manner. Typically, g is a primitive root modulo M. g is a public parameter.

u_k: The public round state for round k. This is an element of Z_M^* (i.e., 1 <= u_k < M). In each
round of encryption, a fresh u_k is used. The sequence {u_k} is deterministic for those who know the
secret trajectory parameters (like step sizes), but appears pseudorandom to others. The initial state
u_0 (for round 1) is provided by the server during the handshake. Subsequent u_k are either transmitted
with each message or recomputed by each side using the shared secret state update rule.

a_k: The public phase for round k, derived from u_k. It is computed as
a_k = (u_k + inv(u_k)) * inv2 (mod M) (Eq2), where inv(x) denotes the modular inverse of x in Z_M,
and inv2 is the modular inverse of 2 mod M. The value a_k lies in Z_M and is used in the encryption
transform. The pair (u_k, a_k) can be viewed as public state parameters for the encryption of that
round’s message. Note that given u_k, anyone can compute a_k easily (so it may not always need to be
explicitly transmitted if u_k is known).

step_u: A secret trajectory parameter controlling how the public state u evolves each round. For
example, a simple state update rule is u_{k+1} = u_k * g^{step_u} (mod M). The integer step_u (which
may itself be randomly chosen or derived from the private key structure) is known to both client and
server after the handshake but is never revealed publicly. Without

knowledge of step_u, an attacker
cannot predict future states from past ones.

door values (b_i): Optional per-layer additive constants injected into the encryption recurrence
to enforce a particular secret structure or order. In some profiles, each layer i of the multi-layer
encryption uses an added term b_i (mod M) derived from the secret decomposition (for example, from
factors of R or their permutation). These values are kept secret (shared only between sender and
receiver) and if used, they modify the encryption formula. The term "door" implies that these values
act as additional one-way doors; without knowing the correct b_i sequence, an attacker cannot properly
encrypt or decrypt. (This use of b_i is distinct from the exponent b in the abR structure.)

stars: A conceptual term referring to internal atomic factors or values derived from the secret
structure which may be used to randomize encryption. For instance, the prime factors of R (after some
processing) can be thought of as "stars". Star values might be used to derive door values or to vary
certain steps of the encryption (such as choosing different q-related multipliers per layer). Some
deployments publish *star projections*, which are transformations of these secret star values (e.g.,
g^(star) mod M), in a shuffled order. The actual usage and ordering of stars in the encryption process
remains secret (determined by parameters like star index and rotation), so an adversary may see a set
of possible contributions but cannot know which were applied when.

rotation: In the context of ASL, rotation refers to a secret offset or scheduling parameter that
determines how internal factors (stars or others) are rotated or selected across encryption layers or
rounds. For example, a rotation parameter ‘rot0‘ might set an initial starting index into a list of star
factors, and with each layer or message, the index shifts according to another secret step (step_star).
Rotation ensures that even if multiple internal factors are known in aggregate (e.g., via published star
projections), the sequence in which they influence the encryption is unpredictable to an attacker. The
rotation parameters are shared secrets between client and server.

round: A single encryption and transmission event in an ASL session. Each round k uses a unique u_k (and corresponding a_k) and results in a ciphertext (or set of ciphertext blocks) that constitutes one message. A session will have an ongoing round counter. Rounds may increment for every message sent in one direction, or they may alternate between parties (depending on the implementation, e.g., client-to-server messages could use odd rounds and server-to-client even rounds). In any case, both sides keep track of round numbers and associated state to remain in sync.

q^2_masked: A public value included in the server’s key profile which serves as an *authenticated commitment to the value of q^2*, without revealing q. This value acts like a static signature or identity binding for the server’s hidden prime. For example, q^2_masked might be computed as a certain residue or combination: such as q^2 modulo some large public number, or q^2 with certain parts obscured by secret factors. The design of q^2_masked is such that an attacker cannot derive q from it, but a legitimate client (who has obtained the correct public key parameters out-of-band or from a trusted directory) can use it to verify that the server’s provided parameters are consistent with the genuine q. In ASL, q^2_masked is used during the handshake to prevent man-in-the-middle impersonation: it MUST match the expected value for the server’s identity, or else the client aborts the connection.

ASL session: A run of the protocol between a client and a server, from the initial handshake through the exchange of data rounds, terminating either by explicit closure or timeout. An ASL session yields a set of shared secrets (trajectory parameters, etc.) and an evolving state, which are used to encrypt and decrypt messages between the parties.

3. The Phi-ns-x Cryptographic Primitive

Phi-ns-x (also referred to as the ns-mod transform) is the sequential encryption core used by ASL. It operates in a multiplicative group of integers modulo a prime M. This section describes the mathematics of phi-ns-x in detail, including the derivation of its equations (Eq0 through Eq6) and

an explanation of its cryptographic properties.

At a high level, phi-ns-x can be seen as a keyed non-linear transformation that takes an element (representing a plaintext or intermediate state) and produces another element (ciphertext or next state) such that reversing this transform is infeasible without knowledge of a secret (the prime p or equivalent internal parameters). Each transformation can be iterated (layered) to amplify confusion and hide any algebraic relationship between input and output. The design is inherently modular and does not rely on any polynomial or linear structures that could be exploited by an attacker.

3.1. Fundamental Equations and Structure

We begin with the fundamental equation defining the relationship between the two primes p and q, and the composite value T:

(Eq0): q^2 - p^2 = T

This equation is the cornerstone of the Phi-Ns cryptosystem. T must be non-zero, positive, and composite. Given Eq0, one can express T as a difference of squares: T = (q - p)*(q + p). However, simply knowing T (if it were published) would trivially leak information about p and q (since one could attempt to factor T or solve the equations for p and q). Therefore, T itself is never revealed. Instead, the structure of T is hidden through factorization and encoding.

The first step in handling T is to factor out small prime powers 2^a and 3^b:

(Eq1): T = 2^a * 3^b * R

where ‘a‘ = ord_2(T) (the exponent of 2 in T), and ‘b‘ = ord_3(T / 2^a) (the exponent of 3 after removing the 2-power). After extracting these, R = T / (2^a * 3^b) is the remaining factor. By definition of a and b, R is not divisible by 2 or 3. R could be prime or composite; if composite, it may have several prime factors which are typically of varying sizes.

The tuple (a, b, R) is the raw "structured secret" derived from T.

All parts of this tuple are kept secret
in Phi-Ns. In practice, the values a and b are relatively small (they count small factors), while R carries
most of the cryptographic weight (bits of entropy).

Next, Phi-Ns employs a **randomized serialization** of (a, b, R) to ensure that no structure of R or the
order of factors leaks. This process can be summarized as:

- Partially factor R (not revealing full factorization if R is large, but extracting small factors up
to a certain bound). Each factor (whether prime or an unfactored cofactor) is considered an "atom".
- Generate a random seed and salt for this key.
- Permute the list of atoms using a pseudorandom permutation derived from the seed.
- Partition the permuted atoms into a certain number of blocks (the number of blocks may itself be derived
from a pseudorandom function of the seed or salt).
- Construct the serialized encoding as: [a, b, salt, block_count, (block1 || block2 || ... || blockN)],
where || denotes concatenation of the binary representation of each block. Each block might be delimited
by length or another indicator so that the parsing is unambiguous.

The serialization is deterministic given the seed, but appears random to anyone without the seed. Even if
an attacker could factor R (which might be extremely large or structured to resist factoring), the permutation
and block partitioning mean the attacker cannot tell which factors correspond to which positions in the original
T structure. In essence, many different tuples (a, b, R) could result in the same multiset of factors, and the
serialization further obscures the arrangement. The total number of possible encodings of the same T (due to
permutations and partitions) is combinatorially large, adding another layer of difficulty to any brute force attack.

In implementations, the final serialized abR is typically committed via a cryptographic hash (e.g., a SHA-256
digest) or another one-way function, and that commitment is published as part of the public key. In ASL, we
avoid external hash functions as a core dependency (per the design goal of pure arithmetic security), so an
alternative approach is used: the q^2_masked value and possibly other arithmetic check values serve to commit
to this structure. Still, conceptually, one can imagine that the client is assured of the integrity of abR

through these mechanisms, meaning the server cannot change its internal structure without detection once the public key is set.

To summarize, the public key information made available (depending on mode) is:
- Either p or q (one prime, not both).
- A commitment to the serialized abR (through q^2_masked or similar means).
- Possibly, parameters like M, g, and some initial state or other auxiliary public values.

The attacker is missing one prime and the entire abR tuple. The security assumption is that finding the missing values is infeasible.

3.2. Sequential Encryption Recurrence (ns-mod)

Phi-ns-x defines how a message is encrypted in a sequential (layered) fashion using the public state parameters (u_k, a_k) and the secret prime (with its abR structure). All operations take place in the finite field Z_M.

Let us consider a single round k of encryption (for a given session state). We have:
- Public inputs: u_k in Z_M^*, and a_k computed as a_k = (u_k + inv(u_k)) * inv2 (mod M). (We restate Eq2 here.)
- A plaintext message represented as an integer m in the range [0, M-1] (if m is larger or is a byte sequence, it can be parsed into one or multiple integers mod M or handled with padding; for now assume m fits in Z_M).
- The encryption will be applied in L layers (iterations) to increase non-linearity. L may be a fixed parameter or derived from the private key (for instance, it could correlate with the depth of factorization or be a security parameter indicating how many times to apply the recurrence).

We denote p_0 = m (the plaintext as the initial state for encryption). Then for each layer i from 1 to L, we compute:

- **(Eq3)**: y_i = (p_{i-1} + a_k) mod M.

This is a simple shift of the previous state by the phase a_k. Note that if p_{i-1} = -a_k (mod M), then y_i = 0; in general encryption we must ensure y_i != 0 (mod M) because we will require a modular inverse of y_i in the next step.

If y_i does turn out to be 0, this indicates a degenerate case for this round (a collision between -a_k and the state);

the specification dictates that in such a case the encryptor SHOULD abandon this round and either increment to the next round (i.e., update u and a to new values) or apply an agreed-upon remediation (such as adding a small nonce to m and trying again). This situation is extremely unlikely for random a_k and p_{i-1} unless an attacker deliberately crafts m = -a_k, but we mention it for completeness.

Given y_i (which is non-zero mod M), define inv(y_i) as the multiplicative inverse of y_i modulo M. The core encryption transform for layer i is then:

- **(Eq4)**:
  ‘‘‘
  p_i = ((y_i^2 - y_i) + (p_{i-1} * a_k)) * inv(y_i) (mod M).
  ‘‘‘

Expanding and simplifying Eq4, we can see the transformation more clearly:

  p_i = [y_i^2 - y_i + p_{i-1} * a_k] * y_i^{-1} (mod M).

This can be interpreted as a quadratic recurrence. If we substitute y_i = p_{i-1} + a_k, then:
y_i^2 - y_i = (p_{i-1} + a_k)^2 - (p_{i-1} + a_k) = p_{i-1}^2 + 2*p_{i-1}*a_k + a_k^2 - p_{i-1} - a_k.
So the numerator in Eq4 becomes: p_{i-1}^2 + 2*p_{i-1}*a_k + a_k^2 - p_{i-1} - a_k + p_{i-1} * a_k.
Simplifying: = p_{i-1}^2 + (2*a_k + a_k)*p_{i-1} + a_k^2 - p_{i-1} - a_k.
= p_{i-1}^2 + 3*a_k * p_{i-1} + a_k^2 - p_{i-1} - a_k.

However, this expansion is not particularly illuminating for cryptanalysis; what matters is that the update from p_{i-1} to p_i is highly non-linear (it involves a quadratic term p_{i-1}^2 and cross-term p_{i-1} * a_k, all under modular arithmetic). The division by y_i (which itself depends on p_{i-1}) further entwines the terms.

After L layers, we obtain p_L, which we denote as X (the result after encryption layers). If L=1, we have a single application of the transform; typically L might be on the order of 2 to 5 for practical parameters, but it can be larger if needed. The final value X = p_L is the output of the sequential encryption for this round.

It’s important to note that the encryptor, to compute this, only needs to know:

- a_k (publicly computable from u_k).
- The value u_k (public, but not directly used in formula except to get a_k).
- The plaintext m (as p_0).
- The number of layers L and if any other per-layer parameters are fixed by protocol.

The encryptor does **not** need to know p, q, or any of the secret decomposition (those are only needed for key generation and possibly for the receiver to verify or derive some things). This is intentional: it means anyone with the public parameters and current state can encrypt a message. In ASL, however, not just anyone should be allowed to encrypt data that the server will accept; we mitigate unauthorized encryption through the use of shared secret parameters (like door values or required structure in the ciphertext) described later. But mathematically, the transform itself is public.

On the decryptor side, if one has the secret prime p (and knowledge of a_k and X), recovering p_{L-1} (and eventually m = p_0) is possible by inverting the recurrence using p and the internal key structure. The inversion is non-trivial for an outsider but straightforward for someone with the correct key: essentially, the decryptor will run a process to find which candidate p_{L-1} could produce X under the transform, guided by the secret knowledge (like knowing q or certain invariants). This process is beyond the scope of this section, but suffice to say it exists and is efficient for the legitimate party. For an attacker, trying to invert without p is akin to solving a random quadratic equation in a large finite field with no contextual cluesa problem believed to be infeasible for cryptographically chosen sizes.

For completeness, we note that some algebraic rearrangement can express Eq4 in alternative forms. For example, one can multiply out the division to get:
p_i * y_i y_i^2 - y_i + p_{i-1} * a_k   (mod M),
or:
p_i * (p_{i-1} + a_k) (p_{i-1} + a_k)^2 - (p_{i-1} + a_k) + p_{i-1} * a_k   (mod M).
Simplifying that:
p_i * (p_{i-1} + a_k) p_{i-1}^2 + a_k^2 + 2*p_{i-1}*a_k - p_{i-1} - a_k + p_{i-1}*a_k   (mod M),
which reduces to:
p_i * (p_{i-1} + a_k) p_{i-1}^2 + 3*p_{i-1}*a_k + a_k^2 - p_{i-1} - a_k (mod M).
There is no obvious further simplification or factorization here; this

is a purposely irreducible-looking relation
mod M, which is good for security.

In summary, the sequential encryption recurrence (Eq3 and Eq4) takes an input p_{i-1} and produces p_i under control of "keyed" value a_k. For a full encryption round of a message, we iterate this L times (using the same a_k each layer in a given round k).

After L layers, the final output X = p_L is either used directly as the ciphertext of round k or undergoes a final post-processing to bind it to the identity parameters (see below). For many scenarios, X itself is transmitted as the ciphertext.

If we make an analogy, one can think of phi-ns-x as a sort of "stream cipher" or "state cipher" where a public state (u_k, a_k) acts somewhat like an IV (initialization vector) and the secret prime p and its hidden structure act like the key that allows decryption. However, unlike a conventional stream cipher IV, here the IV (state) changes in a correlated way each round under secret control (step parameters), and unlike a normal block cipher, the transformation is heavily non-linear with a trapdoor.

3.3. Optional Layer Injection and Masking

The base transform described by Eq3 and Eq4 is already non-linear. However, to further thwart any attempt at cryptanalysis, some deployments of phi-ns-x introduce additional secret inputs per layer. We refer to this as "door injection" because it introduces extra secret "doors" that one must know to successfully navigate the encryption path.

Concretely, if we have secret door values b_i for layer i (with 1 <= i <= L), the encryption recurrence can be modified to:

- **(Eq5)**:
  '''
  p_i = ((y_i^2 - y_i + p_{i-1} * a_k) * inv(y_i) + b_i) (mod M).
  '''

Compare Eq5 with Eq4: we simply add b_i (mod M) to the result of the core transform. The value b_i is chosen in some deterministic but secret way, possibly derived from the abR structure. For example, if R has multiple prime factors,

each layer’s b_i could be linked to one of those factors or some function of it. Another approach is to derive all b_i from a pseudorandom generator seeded with some secret (like t0 in the blockchain example, which might be a random or chosen starting value).

If door values are used, the precise method by which they are generated and applied MUST be agreed upon as part of the protocol profile. This document specifies the extension point but not the exact derivation, as it can differ based on use case:
- In one profile, b_i might all be zero (meaning no injection, which defaults to Eq4 behavior).
- In another, b_i might cycle through a small set of constants derived from the secret key.
- In an advanced usage, b_i might be tied to "star factors": e.g., if R had factors r1, r2, r3..., we might define a series where b_1 = f(r_{idx0}), b_2 = f(r_{idx0+1 mod n}), etc., effectively rotating through factors with some function f (like taking a reduced residue). The ‘rot0‘ and ‘step_star‘ parameters (see Terminology) would control how the index through star factors advances with each layer or round.

The effect of Eq5 is that even if an attacker somehow deduced the functional form of Eq4, they would still be missing an additive piece in each layer. Without knowing b_i, an attackers attempt to guess p_i from p_{i-1} and a_k would systematically fail, as they’d always be off by this unknown offset. Only the true receiver, who can either compute or brute-force check the correct b_i (knowing the key, there might be only one value that makes the decrypted plaintext intelligible or fits the key structure), can properly invert the equation.

For interoperability, ASL defines that by default no layer injection is used (b_i = 0 for all i) unless a specific profile explicitly incorporates it. This keeps the core specification simpler. However, implementations SHOULD support at least the possibility of one non-zero door value (e.g., a constant offset or a simple pattern) to be future-proof or to integrate with extended profiles that enhance security via this mechanism.

Additionally, beyond per-layer injection, one may consider a final masking of the ciphertext. One such method, inspired by the Phi-Ns identity binding, is to incorporate q (or T) into the output. For instance:

- **(Eq6)**:
  ‘‘‘
  C = (X + q)^2 - T * a_k^2,
  ‘‘‘
  where X = p_L is the output after L layers. In this construction, the transmitted ciphertext C is a quadratic
  residue that depends on q and T. The receiver, knowing T (since they know p and q), can compute (X + q)^2 = C + T * a_k^2,
  and since they know q, they can solve for X (because (X+q)^2 is known, taking square root yields two options (X+q),
  but one of those will be consistent with the expected sign or range of X+q, etc.). This effectively uses the knowledge
  of T as an extra key: an attacker who intercepts C cannot recover X without solving the equation involving T (which
  is secret). If M and the size of q are large enough, even trying to brute force q from q^2 (mod something) would be
  infeasible, especially since a_k^2 * T is subtracted which acts as a blinding factor for outsiders.

The Eq6 example is just one possible "ciphertext projection". ASL may employ simpler approaches for final output as well,
such as transmitting X as-is, or X plus some constant mask. The choice can be left to profile definitions. The goal of any
such masking is to tie the ciphertext to the static key in a way that doesn’t affect legitimate decryption but makes
cryptanalysis harder.

3.4. Cryptographic Properties of Phi-ns-x

The phi-ns-x transform possesses several properties that underpin the security of ASL:

- **Non-linearity**: The core equations (Eq3Eq5) are non-linear (quadratic) in the plaintext/state variable. Unlike
  linear transformations (which could be solved or inverted with linear algebra or by observing linear relations),
  phi-ns-x produces outputs that are quadratic residues of input combinations. The presence of terms like p_{i-1}^2
  and p_{i-1} * a_k inside a modulus means the transformation does not have a simple algebraic inverse without knowing
  secret parameters. Cryptanalytic attacks that rely on linearity, such as certain forms of differential or linear
  cryptanalysis, find no straightforward linear approximation here.

- **High Entropy Mixing**: Each layer of transformation mixes the previous state with the phase a_k and (if used) door
  value b_i. Even if the plaintext has a structure or the attacker can choose plaintexts, the output after one layer is

significantly de-correlated from the input. After L layers, the output is effectively psuedo-random from the attacker’s perspective (assuming a_k is not under attacker controlwhich it isn’t, since it’s derived from a secret-driven state u_k). The introduction of multiple layers and optional per-layer secrets ensures that even a single round of encryption in ASL provides a high-entropy ciphertext.

- **Combinatorial State Space**: If an attacker attempts to guess the secret (whether it’s p or the abR structure or the door values), they face a combinatorial explosion. For example, consider the problem of forging a valid ciphertext without knowing the secrets. The attacker would have to guess the correct p_{0} (plaintext) that, under the unknown transform, yields a meaningful result, guess the unknown b_i for each layer if any, and produce an output that the receiver will accept. Because any wrong guess is effectively undetectable to the attacker (there is no oracle giving feedback if a guess is "close" or partially correct), the attacker is left with an all-or-nothing brute force. This is akin to the situation with the underlying Phi-Ns key: an attacker guessing p gets no feedback that helps refine the next guess.

- **No Direct Factorizability**: The encryption transform does not lend itself to being "factored" or decomposed into independent sub-problems. In some ciphers, an attacker can break the problem into parts (for instance, attacking rounds one by one, or separating a key into two halves via meet-in-the-middle). In phi-ns-x, the structure is such that any partial knowledge (say an attacker hypothetically guesses part of the state or one layer’s output) does not linearly reduce the complexity of determining the rest. The presence of non-linearity and modulus coupling means that solving for any unknown inherently requires solving the whole set of equations. This holistic complexity is by design; it’s analogous to the way that in Phi-Ns key recovery, one must guess everything (p and full abR) to verify a solution, with no way to systematically solve piecewise.

- **Entropy Amplification via Recursion**: If recursive decomposition is used in key generation, the effect on security is an exponential increase in search space for an attacker. For the legitimate participants, recursion is mostly transparent (they just have a larger key to work with, but encryption/decryption operations remain the same top-level ones). For an attacker, however, recursion means that even if they tried to brute force something like p or q, that

would not be sufficientthey would face another hidden layer of structure inside p itself, and so on. The security of ASL with recursively structured keys thus compounds with each layer of recursion.

- **Post-Quantum Resistance**: There are no known quantum algorithms that can solve the Phi-Ns inversion problem in polynomial time. Shor’s algorithm, which breaks RSA and ECC, is ineffective here because it requires working with a known composite or performing an order-finding in a group derived from known primesPhi-Ns reveals no such composite and no such group structure to the attacker. The best one could do with a quantum computer is apply Grover’s algorithm to brute force search for the secret, which gives a quadratic speedup. However, because the search space (the set of possible p and abR combinations consistent with the public info) grows combinatorially with key size (especially due to randomization and recursion), even Grover’s quadratic speedup is insufficient to threaten realistic parameter sizes. For example, if the classical brute force space is 2^256, Grover might reduce that to 2^128 steps, which is still computationally infeasible.

- **No Partial Disclosure or Oracle**: The design of ASL and phi-ns-x ensures that an attacker cannot gain partial information that leads to a snowball effect. The protocol never discloses T, the exact abR ordering, or any one-way function output that could be inverted. There is also no encryption or decryption oracle accessible to attackers: during handshake, if an attacker tries to impersonate or send incorrect values, the protocol will simply fail (the client or server will abort), without giving the attacker clues as to which part was wrong. During data transmission, an attacker cannot query the decryptor in any useful wayif they send a bogus ciphertext, the receiver will either drop it (if authenticity checks fail) or decrypt to nonsense. This "no oracle" principle is inherited from Phi-Ns and is preserved in ASL. It is crucial for preventing adaptive attacks; an adversary cannot, for instance, tweak a ciphertext slightly and ask "did that decrypt correctly?" as a means to learn about the key.

In summary, the phi-ns-x primitive provides a robust cryptographic core for ASL. It ensures that without the private key material (p and abR, plus any shared trajectory secrets), an attacker faces a cryptographically hard problem to either decrypt messages or forge new ones. The combination of structural secrecy (from Phi-Ns) and dynamic state secrecy

(from phi-ns-x’s state evolution) makes ASL a moving target, cryptanalytically speaking.

4. ASL Protocol Specification

In this section we describe the ASL protocol in detail. We start with the roles and the components that each party holds, then delineate the handshake (session establishment phase), followed by the ongoing message encryption process, and the exact format and encoding of messages.

4.1. Roles and Key Components

ASL involves two primary roles: the **client** and the **server**. Typically, the server is the party that has a long-term key pair (the Phi-Ns key) and provides a service (like a website, an API endpoint, or a blockchain node), and the client is the party initiating the secure session, who may or may not have its own long-term key. This asymmetry mirrors usage patterns of TLS (servers present certificates, clients initiate handshake). However, ASL can be used in a peer-to-peer context as well, where both parties have keys and either can act as "server" for initiating a connection.

Server Key Materials: The server possesses:
- A key pair (p, q) along with the secret decomposition (a, b, R). Depending on the chosen mode:
  - In PK-q mode: q is the public key, and p with abR is private.
  - In PK-p mode: p is public, and q with abR is private.
- A set of public parameters associated with the key:
  - The prime modulus M used for phi-ns-x operations.
  - The generator g for Z_M^*.
  - Possibly one or more recommended profiles or parameter sets (like the typical size or number of layers L).
- A commitment to the secret structure:
  - Either a cryptographic hash of the serialized abR, or in our design, a numeric value that serves the same purpose (q^2_masked).
- Optionally, a set of *auxiliary public values* that assist in encryption without revealing secrets:
  - For PK-p mode: values derived from q or R that allow the client to perform encryption. For example, **q_anchor** (an "anchor" value related to q, such as q mod M or q mod some smaller base) and **star projections** (a list of g^f mod M for some secret factors f). These were illustrated in the blockchain use-case as ‘q_anchor‘ and a list of star projections. In a general protocol context, these might or might not be transmitted, but if used, they are included in the server’s initial handshake message.
- Policy information: e.g., how often the key is refreshed (the Phi-Ns

key can be re-randomized via abR re-serialization without changing p or q), supported modes (PK-p or PK-q or both), and supported use of advanced features like door injection.

The server’s **public key profile** can thus be summarized as a structure containing: { mode, p_or_q, M, g, [q^2_masked], [q_anchor], [star_projections], [others...] }. This profile would be published or made available to clients via some secure channel (like DNS, a blockchain, or an offline distribution), similar to how a TLS server has a certificate.

Client Key Materials: In many scenarios, the client may operate without a long-term key pair (just as most web clients do not have a certificate). ASL can accommodate a client with no long-term keys; the client will simply generate some ephemeral values during the handshake (like a random session secret to send to server). However, ASL also allows mutual authentication: a client could have its own Phi-Ns key pair and authenticate itself to the server in a similar way (this would be analogous to client certificates in mutual TLS, but achieved via the same mechanism as the servers authentication). For brevity, we focus on the typical case of server-authenticated, client-unauthenticated sessions, but we will note where a client key could come into play.

Regardless of having a long-term key, the client will generate some **ephemeral secrets** during the handshake:
- A random session secret ‘K_s‘ (for example, 256 bits) that will be used to derive shared parameters (like step_u, step_star, etc.). Instead of directly using this as a symmetric key as in RSA-based TLS, ASL can incorporate this secret into the phi-ns-x encryption so that only the server (with its private key) can recover it.
- Potentially an ephemeral Phi-Ns key pair for forward secrecy: The client might generate a fresh small Phi-Ns instance (p_e, q_e) just for this session and provide q_e in the handshake (with a commitment to ensure authenticity). The server could then combine keys in some manner or simply switch roles and encrypt something back to the client using the client’s ephemeral public key. This is an advanced option and not required; forward secrecy can also be achieved by discarding the session secrets after use.

After the handshake, both client and server will share:
- The trajectory parameters for state updates: e.g., step_u (for u sequence), step_star (for star rotation), rot0 (initial rotation), star_idx0 (starting index for star usage), L (number of layers for phi-ns-x), etc. These are either negotiated or derived from the exchanged secret.
- The current state values: u_0 (initial state for round1, provided by server), and consequently a_0 (which client can compute).
- Any symmetric key material needed for additional encryption or

authentication (if we overlay a MAC or use symmetric encryption for payload, but ASL’s design tries to avoid separate MAC by making ciphertext self-authenticatingstill, a session might derive a key to, say, authenticate certain out-of-band signals or to mix into application protocols).
- Optionally, if client authentication was performed, the server similarly learns some shared secret that confirms the client’s identity.

In summary, the server is the holder of the master key (Phi-Ns key) and provides public parameters; the client initiates communication and contributes randomness to create unique session keys. The protocol ensures that after the handshake, both sides have identical sets of secret parameters which they will use to synchronize the stateful encryption.

4.2. Session Establishment (Handshake)

The ASL handshake is the process by which the client and server establish a secure context. The handshake achieves:
1. Server Authentication: The client gains confidence that it is communicating with the legitimate server (not an impersonator), via the server demonstrating knowledge of its secret key (p, q) in a non-forgeable way (using q^2_masked and successful decryption of a test secret).
2. Establishment of Shared Secrets: The client and server arrive at common values (the "session keys" or parameters) that will be used for encrypting and decrypting application data. Only the two parties learn these values.
3. (Optional) Client Authentication: If required, the server can also authenticate the client, either via a similar mechanism or by verifying that the client knows a pre-shared secret. In many applications this step is omitted, or handled at the application layer (e.g., via a login after the secure channel is up).

The handshake can be described as a sequence of message exchanges. We define the messages in an abstract format here:

- **ClientHello**: This is the first message sent by the client to the server. It may include:
  - A protocol version identifier (to handle future versions or backward compatibility).
  - A list of cipher suite preferences or capabilities (in ASL’s context, this could indicate whether the client supports certain profiles, e.g., injection or recursion or PK-p vs PK-q mode preferences).
  - A random nonce or session identifier (client_random) for the handshake, to ensure freshness.
  - Optionally, an ephemeral public key of the client if doing mutual

key exchange (e.g., the client could include q_e (ephemeral) and a commitment of its own, but this is optional).
  - Optionally, any cookie or anti-DoS token if a prior stateless negotiation mechanism is used (similar to DTLS or IKE cookies; this is an operational detail not core to cryptography).
  - The ClientHello in ASL can actually be minimal if the protocol doesnt need negotiation: in the simplest case, it might just be a packet indicating "I want to start ASL handshake" plus a random.

- **ServerHello (ServerParams)**: The server responds with its parameters. This is a critical message containing:
  - The selection of protocol version and profile (if any negotiation; likely just mirrors what client said if supported).
  - The server’s public key information:
    - ‘mode‘ (PK-p or PK-q).
    - ‘p_public‘ (if mode is PK-p) or ‘q_public‘ (if mode is PK-q). We’ll use p_public to denote whichever prime is being revealed as public.
    - ‘M‘ (modulus for encryption).
    - ‘g‘ (generator for state updates).
    - ‘q^2_masked‘ (the masked commitment to q^2, as defined in Terminology).
    - If mode is PK-p (so q is secret), then **auxiliary encryption parameters** must be provided so the client can perform encryption. This could include:
      - ‘q_anchor‘: a value that serves as a temporary anchor for encryption calculations. For example, the server might provide q mod M as q_anchor. Since p is public, q mod M alone might not give away q if M is much smaller than q or if multiple possible q yield the same mod M (which can happen if q > M).
      - A set of ‘star_projections‘: e.g., a shuffled list of g^(factor) mod M for factors of R or related values. This effectively gives the client a pool of values that relate to secret components, one of which might need to be used in encryption layers, but the attacker doesn’t know which or how.
      - The rationale is that with p_public known, the client can compute T mod M if it had q mod M (because T = q^2 - p^2, so T mod M = (q mod M)^2 - (p mod M)^2 mod M). If M is large and not a factor of T or anything, this by itself may not leak too much. Anyway, the specifics of q_anchor usage would be profile-defined. The point is the server supplies whatever is necessary for the client to execute phi-ns-x encryption.
    - If mode is PK-q (so q is public), then providing q itself is enough for encryption; auxiliary parameters might not be needed, though the server could still provide some precomputed values or recommendations (like how many layers to use).
  - The server’s initial state:
    - ‘u_0‘ (or u_1 depending on indexing) for the first round of

encryption after the handshake.
    - The server might also supply ‘a_0‘ for convenience, but since a_0 can be derived from u_0, this is optional. However, including a_0 allows the client to double-check calculation.
    - A round number or session ID if needed. For example, round=0 might be implicit for handshake phase.
  - A server_random nonce and possibly a session ID (to pair with client_random for key derivation and to avoid replay).
  - Optionally, a signature or proof over all these parameters. In classical TLS, the server would now send a Certificate and a signed ServerKeyExchange to prove possession of the private key corresponding to the certificate’s public key. In ASL, we achieve this proof not by an external signature algorithm but inherently:
      * The inclusion of q^2_masked is one form of static proof (the client will verify this matches the known value for this server identity).
      * Additionally, the server will prove its private key knowledge by being able to decrypt the client’s next message correctly. If the server were fake and didn’t have the key, it would fail to respond properly in the next step, which the client would detect.
      * However, there is a potential for a subtle MITM if we aren’t careful: an attacker could relay messages between client and real server without knowing the key. To prevent that, some binding of handshake messages is needed, similar to the Finished message in TLS. We will address that with a final handshake verification.
  - (If client authentication is desired) a request for client key or info, akin to a CertificateRequest in TLS. In ASL, that could be an indication like "please prove who you are by sending an encrypted blob I can verify," but we’ll not elaborate here.

- **ClientKeyExchange (ClientEncryptedSecret)**: After receiving the server’s parameters, the client verifies:
  - The server’s ‘p_public‘ or ‘q_public‘ matches the expected identity (for example, the user might have the server’s p or q fingerprint configured, or in a web scenario, this would be checked against some known value or a trust on first use store).
  - The ‘q^2_masked‘ matches the known commitment for that server (this might be published through DNS or a public ledger or given out-of-band). If this check fails, the client MUST abort the handshake (since it indicates the server’s parameters are not consistent with the legitimate key).
  - If the check passes, the client proceeds to encrypt a secret for the server. The client generates the random session secret ‘K_s‘. This can be, for instance, 32 bytes of cryptographically secure random data. The client then encrypts ‘K_s‘ using the phi-ns-x transform under the provided state and parameters:
    * The client takes the plaintext m = an encoding of ‘K_s‘ (if ‘K_s‘ is 256 bits, interpret it as a number modulo M; if its larger, it might

be split across multiple messages or handled by chunking).
    * The client uses the round state provided (‘u_0, a_0‘). Let’s call this round "handshake round" or round 0. It will apply the recurrence Eq3 and Eq4 (and Eq5 if doors are used) L_h times, where L_h might be fixed or chosen by servers profile. The output will be X_secret.
    * If PK-p mode is used and the server supplied auxiliary parameters like star projections, the client will incorporate them as specified. For example, if a door value b_i is derived from a star factor and the server gave a list of star projections, the client might not know which to use. However, the safest approach for handshake is not to require the client to guess any secret injection. So the server could specify that for the handshake round, no injection is used (or a fixed simple pattern) just to convey the secret. Alternatively, the server could supply an explicit hint: e.g., an index or one of the star projection values that the client should use for encryption in this handshake. To keep things straightforward, assume that the handshake uses a profile with no injection (or the server picks parameters such that b_i=0 for this round). The primary goal is to get ‘K_s‘ across.
    * The client sends **ClientEncryptedSecret** = X_secret (the ciphertext). Additionally, the client may send computed verification data:
      - It can compute a hash or MAC of the handshake messages (ClientHello + ServerHello + X_secret) using ‘K_s‘ and include this as well (similar to how TLS’s Finished message is a MAC over all prior handshake messages using the new keys). This serves as a strong confirmation that the handshake hasn’t been tampered with. This can be done by, say, taking ‘HMAC(K_s, handshake_transcript)‘ or a straightforward hash since we want to minimize external primitives. If we avoid hash functions entirely, we could use phi-ns-x itself in a specialized way to produce a tag (e.g., encrypting a constant with a known pattern using ‘K_s‘ as input could produce a tag). But including a small SHA-256 here purely as a handshake integrity check is not against the spirit of ASL, since it’s not used for core security, but let’s assume we want to avoid even that. Another way is: The server, when it decrypts ‘K_s‘, could produce a known response that implicitly confirms the handshake integrity.
      - To keep the initial specification free of additional primitives, we will proceed without an explicit Finished MAC, but note that this is a point where a profile could add one for enhanced security.

- **ServerFinish (Handshake Completion)**: Once the server receives the ClientEncryptedSecret, it attempts to decrypt it:
  - The server knows p (or q in PK-q mode) and the full secret key, so it can run the inverse of phi-ns-x on X_secret using the same round state (u_0, a_0) and (if applicable) the known door values. This yields a candidate plaintext which should be ‘K_s‘.
  - If the server fails to decrypt properly (e.g., because the attacker tampered with X_secret or it was encrypted under wrong assumptions), the

result will be gibberish. The server can detect failure perhaps by format (if ‘K_s‘ has some structure or length). If decryption fails, the server MUST abort the connection.

- If decryption succeeds and yields a plausible ‘K_s‘, the server now has the shared session secret.
- At this point, both sides have ‘K_s‘. They can derive from it the internal parameters needed for the ongoing encrypted session. A simple derivation could be:
    * step_u = H1(K_s, client_random, server_random) mod (some range) where H1 is a hash or KDF function. But again, to avoid hash, they might directly treat ‘K_s‘ bits as comprising these values. For example, split ‘K_s‘ into two halves: first half as step_u, second half as step_star (just as bits of an integer). If ‘K_s‘ is uniformly random, each half is random. Or derive rot0 from a subset of bits, etc. The exact method should be agreed: the point is, ‘K_s‘ is now the master secret for the session from which all needed shared secrets are determined.
    * For instance: interpret ‘K_s‘ as a 256-bit number. Break it into: step_u (e.g., 128-bit), step_star (64-bit), rot0 (16-bit), and an authentication token (48-bit) these bit lengths are arbitrary examples. The values may be reduced modulo appropriate ranges (like if step_u must be taken mod (M-1) maybe, since it might effectively act as an exponent).
    * Both client and server do the same derivation, so they have identical parameters.
- The server should now send a final confirmation. If we had not included a Finished MAC from client, the server will include something to assure the client that it too derived the same keys and that the handshake wasn’t tampered with. The simplest way:
    * The server sends a **ServerFinish** message encrypted with the newly established session keys. For example, the server can compose a message "HS_DONE" (handshake done indicator) and encrypt it as the first ASL data round (round 1).
    * Concretely, server sets round=1, computes new u_1 from u_0 (with secret step_u), computes a_1, encrypts a known ASCII string or structured data (like a constant or perhaps the handshake transcript hash) using phi-ns-x with L layers under a_1 (and door values etc. now known to both from K_s), producing X_verify, and sends it along with u_1, a_1.
    * The client, upon receiving this, will decrypt using its copy of the secrets. If the plaintext matches the expected value (e.g., "HS_DONE" or matches a hash of handshake messages), then the client knows the server was indeed present in the handshake and that the shared secrets match. At this point the handshake is complete and the channel is secure and authenticated.
    * If the decryption fails or the value is incorrect, the client must terminate the session because it indicates something went wrong (keys mismatch or MITM).
- If we wanted to avoid even one round trip for finish, the clients

previous message could have included something that server would echo back. But it’s cleaner and more symmetric to have this final step.

- After these exchanges, the handshake is complete. The protocol transitions to the data transfer phase.

Summary of Handshake Flow:
Client -> Server: ClientHello (random_C, cipher_suites, maybe client_ephemeral_pub)
Server -> Client: ServerHello (p_public or q_public, M, g, q^2_masked, u_0, [q_anchor, stars...], random_S)
Client -> Server: ClientEncryptedSecret (X_secret = Enc(m=K_s), maybe Finished MAC)
Server -> Client: ServerFinish (Encrypted confirmation with round 1 state)
(Messages could be combined or split differently in implementations, but this is the logical content.)

Throughout this process, no out-of-band trust is needed except that the client must have an authentic reference for
the server’s identity (particularly the expected q^2_masked or the servers public key itself). This could be obtained
via a one-time secure channel or a trusted directory. If that is not available (trust on first use scenario), the
protocol still encrypts the traffic, but a MITM could impersonate the server on the first connection. However, because
q^2_masked would change if they later see the real server’s key, a client could detect it on subsequent connections
(similar to SSH’s known host keys). This is a deployment consideration.

For clarity, let’s tie back to the earlier blockchain example terms:
- ‘p_public‘ in handshake corresponds to that p_public.
- ‘q_anchor‘ and ‘q^2_masked‘ were given in "genesis" in that example; here ServerHello carries them.
- The shared off-chain secrets like step_u, step_star, rot0 are essentially derived from K_s in our handshake.
- The round state (u_k, a_k) in each transaction corresponds to our per-message state below.

4.3. Round-Based Encrypted Messages

Once the handshake has completed, client and server enter the secure data phase. The key aspect of ASL’s data encryption
is that it is stateful: each message is tied to a round index and the public state (u_k, a_k) for that round.

The participants now share:
- The secret trajectory parameters: step_u, step_star, etc., derived

from K_s.
- The starting state: let’s call it (u_1, a_1) which should have been either provided in the ServerFinish or computed
  by each side. Actually, in the handshake above, the server already sent u_0 for the handshake encryption. After using
  u_0, both sides can compute u_1 = u_0 * g^step_u (mod M). If the server already did this and sent something with u_1,
  then the client upon verifying ServerFinish knows u_1 and a_1.
  Alternatively, some might prefer that the server doesnt send u_1 but that both sides compute it independently from
  u_0 and step_u. However, if any message was lost or if one side increments and the other doesn’t, there could be desync.
  Its safer to include u_k in each message to avoid synchronization issues. This is slightly less efficient (sends some
  extra bytes), but ensures robustness.
- The internal secret factors scheduling (like the sequence of door values b_i if used, or star usage pattern): these are
  agreed or known from handshake secrets.

Data is sent in **records** or **messages**. Each record is self-contained in the sense that it carries the information
needed for the receiver to process it (or the receiver can derive that info if in sync).

For each message the following occurs (assuming a message is one protocol data unit to be encrypted and sent; if a
higher-level application message is large, it could be fragmented into multiple ASL messages, each with its own round):
1. The sender (could be client or server) determines the next round number and computes/obtains the corresponding
   state (u_k, a_k).
   - If the sender was the one who sent the last message (i.e., we might not alternate; consider an HTTP response with
     multiple packets), then the sender would have already incremented the state internally after sending. Typically,
     we can define that every message increments the round number by 1, regardless of who sends it. This means both
     sides advance a shared counter even on their own send. That is unusual (because normally each side would increment
     their own sequence), but since state is shared, it is more like an alternating sequence in lockstep.
   - Another approach: maintain two independent sequences, one for each direction (like TLS has independent sequence
     numbers per direction). We could do that: a client_send_state and server_send_state each with their own step parameters.
     That would require deriving separate step_u_client and step_u_server from K_s. This might be more elegant to avoid
     issues of one side not sending for a while and the other wanting to

send many messages (like a server pushing data
     unidirectionally). Yes, it’s probably practical to have separate
state threads so that one side sending many messages
     doesn’t starve the sequence for the other.
   - Let’s assume we have separate sequences: one keyed for client
transmissions, one for server transmissions. In that
     case, we derive step_u_C and step_u_S from K_s. The initial u_C1
and u_S1 can both start at some common value or be
     derived differently (maybe the server provided u_1 in handshake
which was for first server message, and for client
     initial state maybe the client picks or both could use same u but
different step yields divergence).
   - To avoid complicating, though, we can also stick to a single
sequence used by both: in a request-response scenario
     it naturally alternates. But consider asynchronous messaging, a
single sequence is simpler because it globally orders
     messages. This is akin to a half-duplex or synchronized full-duplex
channel. There is a risk if both try to send at
     the exact same round number simultaneously. That could be resolved
by one of them detecting a collision and adjusting
     (e.g., if two messages come with same round from both sides, the
higher party ID wins and the other can resend with
     increment). This is too complex; separate sequences is easier for
practicality.
   - For specification clarity, let’s define:
     * Each side maintains a send sequence (with its own u_k evolving
via step_u known to that side).
     * Actually, step_u is shared, so if both used it they’d produce the
same u sequence which is not what we want if
       they are sending concurrently because they’d collide. If we want
separate, we could have step_u_C = step_u (from K_s)
       and step_u_S = -step_u or another independent value. Perhaps we
could derive two seeds from K_s so that one
       sequence might use g^step_u, the other uses g^step_u2 or maybe
g^(-step_u) to differentiate.
     * For simplicity in this version, we assume a synchronous
communication: client sends a request (round 1), server
       responds (round 2), client next (round 3), etc. This fits
scenarios like request-response or blockchain where
       transactions are sequential by design. If a use case needs
simultaneous asynchronous sending, the protocol can
       be extended to label messages with which sequence they belong to
(C or S) and each with its own state update rule.

2. The sender prepares the plaintext for encryption:
   - This plaintext could be a JSON, a binary blob, etc. It may be
necessary to encode a length or ensure it’s smaller
     than M. Usually M would be chosen large enough to send a reasonable

chunk or a symmetric cipher key. But for streaming,
if data is larger than M can handle at once, one would chunk it.
ASL can either treat each chunk as an independent
message or incorporate a chaining at the phi-ns-x level (the latter
is not typical; easier to chunk).
- If needed, the plaintext is padded to a certain length (maybe to
nearest multiple of some bytes) to avoid leaking
exact lengths. But that’s application specific. Not mandated here,
though adding padding for security (to not reveal
message length patterns) is recommended.

3. The sender encrypts using phi-ns-x:
- They know the current rounds a_k. They set p_0 = plaintext
(converted to int mod M by some agreed reversible scheme,
e.g., big-endian byte interpretation).
- They run the recurrence for L layers (in normal data, L might be
set to a value possibly different from handshake L).
- If door values (b_i) are being used for data encryption, the sender
applies them as per Eq5. Both sides know the sequence
of b_i to use. If b_i sequence is static for all rounds, fine.
Possibly, the b_i sequence could actually depend on the
round number, e.g., using different segments of the star list for
different rounds. That could be orchestrated by using
rot0 and step_star: e.g., at round k, start injecting factors from
index = (rot0 + k*step_star) mod N. This is a plausible
design. It means each round’s multi-layer encryption uses a
different subset of Rs factors, making patterns between rounds
even less correlated.
- After computing X = p_L, the sender may apply final masking if the
profile calls for it. However, typically for sending
data, we might choose to not use an additional mask (or if we do,
both sides must know how to remove it). If something like
Eq6 was used, the receiver would need to know it and apply inverse.
For now, assume the ciphertext is X (mod M).

4. The sender transmits the message which includes:
- The round identifier (k). If we strictly alternate, k might be
implicit (each side expects the next number). In unpredictable
async, better to include it explicitly.
- The public state values for that round: u_k and/or a_k. (One of
these could be derived from the other. Possibly sending only
u_k is enough since a_k can be recomputed. But sending both could
be a nice integrity check: since a_k = (u_k + inv(u_k))*inv2 mod M,
the receiver can recompute and verify the given a_k matches given
u_k. This could catch any corruption or tampering in transit.
An attacker who tries to change u_k would also have to adjust a_k
consistently to avoid detection.)
- The ciphertext X (or C if masked form).

- Any additional metadata, e.g., a flag for last message, or an application-level content type if needed (like TLS records have type).

5. The receiver on receiving a message:
   - Checks the round number (if out-of-order or repeats, might indicate replay or loss).
   - Verifies the state consistency: e.g., if a_k is provided, check a_k ?= (u_k + inv(u_k))*inv2 mod M. If it doesn't match, the message is invalid and should be dropped.
   - If the protocol expects sequential rounds and the round number is not the expected next (and not yet seen), it might indicate either reordering or an attack. Depending on design, might queue out-of-order or just close connection due to suspicion.
   - Using the locally stored secrets:
     * The receiver knows step_u (or two if separate directions, then picks appropriate one).
     * If the receiver has been computing states on its own (for its sending), it might also compute what u_k should be for the peer's message if in sync.
       If they are using separate sequences, the receiver can compute expected u for a given round if it knows how the other side's step works. Actually, if
       separate, maybe the receiver isn't computing the other side's states on its own, instead it relies on what's in the message. And then it updates some
       record of "last seen u from that side".
     * In either case, since u_k is included, the receiver doesn't strictly need to pre-compute it, but it can verify the progression if it has memory of
       previous u from that sender. For example, if last server message had u_x, the client can verify that u_k = u_x * g^{step_u_S}^{(k-x)} mod M (i.e.,
       check it fits the expected sequence progression). If not, either messages were lost or altered. If lost, maybe we can still accept if within a window
       and adjust? But let's not complicate; ideally no loss or the protocol might drop out-of-sync.
   - The receiver then computes a_k from u_k (if not already given or to cross-check).
   - It sets up the decryption recurrence. Now, decryption of phi-ns-x is not simply running the inverse function easily because we don't have a straightforward
     algebraic inverse. But the receiver has the secret p and possibly other private key info. They need to solve p_L = X for p_0.
     * One approach: Because the legitimate receiver has full knowledge of p (the secret prime) and the internal structure, it could simulate the encryption
       forward to see if it matches X, but that would require guess of

plaintext. Instead, likely the receiver uses a more direct method. For phi-ns-x, given
X = p_L, one can iterate backwards:
- There is a relation linking p_L and p_{L-1}: from Eq4 rearranged, knowing p_L (and therefore presumably knowing y_L since y_L = something with p_{L-1}),
it’s tricky to go backward directly without knowing p_{L-1}. However, in the context of the full key, the secret prime p might allow computing some
invariant that helps invert layer by layer. This part is complex; presumably, the design of phi-ns-x ensures decryptability by an algorithm that leverages
p and abR. For example, there might be a way to detect the correct y_i at each step by using the fact that the output must align with the prime’s identity
structure (something like p_i must be a quadratic residue related to p or q in a known way).
- In any case, since phi-ns-x was introduced as an "encryption" transform with the statement that decryptor possesses p and trajectory parameters to decrypt,
we trust that a decryption algorithm exists. Perhaps the decryptor tries all possible factors as candidate b_i inverses if door used, etc. Or maybe since it
knows q, it can compute what y_i should be: e.g., maybe q acts like an eigen or fixed point parameter such that at final layer L, (X + q*a_k)^? yields something
that reveals p_{L-1}. Hypothetically, if we plug p_i = q (just thinking), then Eq4 would simplify. Actually, maybe the trick: The server can test candidates
for p_{L-1} by solving the quadratic equation given p_L and known a_k and known b_L if any. That quadratic is:
‘p_L * y_L y_L^2 - y_L + p_{L-1}*a_k - b_L (if b used)‘ which is essentially ‘p_L * (p_{L-1}+a_k) (p_{L-1}+a_k)^2 - (p_{L-1}+a_k) + p_{L-1}*a_k - b_L‘.
This can be rearranged to a polynomial in p_{L-1}. Likely a quadratic in p_{L-1} as well. The decryptor can solve that quadratic (which might have 0,1, or 2 solutions mod M).
One of those solutions should be the correct p_{L-1}. How to choose the right one? Possibly because only one will continue to yield a valid plaintext in subsequent backward steps or
within range of message size. Then it can continue iteratively. This is plausible as a decryption method.
* We do not need to detail the algorithm in the specification; it’s enough to state that the receiver uses its private key to invert the phi-ns-x layers and recover plaintext.
This may involve brute-forcing a small number of possibilities at each layer (since the equation to invert is quadratic, at most 2 solutions, maybe more with door values considered,

but if star injection made ambiguous, the correct combination must be found).
* We guarantee that given the design of abR and injection, the correct plaintext yields a coherent solution chain that matches the private key structure, whereas a wrong guess quickly
leads to a contradiction (like an impossible factor or mismatched q^2 when check against q).
- After obtaining p_0 (plaintext as an integer), the receiver converts it back to the original message format (removing padding if added).
- The receiver then uses the plaintext or passes it up to the application.

6. State update: After processing a message, both sender and receiver update their expected state for that sequence:
- If using one global sequence, they both increment the round counter.
- If separate, the sender obviously updated its own state when sending. The receiver upon successfully decrypting might recompute what the next expected u for that sender would be (though it can also just wait to receive it in the next message).
- Additionally, if star rotation is used across rounds, both sides update the star index (e.g., star_idx = star_idx0 + round * step_star mod N, or some such formula), so that the next message will use a different starting point in the star list for door values.

Notably, there is no separate "MAC" in each record as one would see in TLS (prior to TLS 1.3) because the encryption itself combined with state verification suffices. If an attacker modifies bits in the ciphertext X, the probability it decrypts to something valid (like meaningful plaintext with correct structure) is negligible; even if it did, the protocol’s state check (u_k and a_k sequence) would likely fail or the content would be nonsense to the application and could be detected with application-level sanity checks. The inclusion of state values provides an implicit integrity check: an attacker cannot easily alter X without also altering a_k and u_k in a consistent way, which would require breaking the encryption anyway.

In practice, an implementation might still want to include an explicit checksum or MAC over the ciphertext and perhaps part of the header (u_k, a_k) using the session symmetric secrets, to guard against any theoretical weaknesses in the transform as an authenticity mechanism. However, we leave this as optional, given the design goals.

4.4. Message Formats and Encoding

We now specify the format of the handshake and data messages in a more concrete manner. These formats are described conceptually;

actual implementations may serialize fields in network byte order, etc. We assume a transport where messages can be delineated (e.g., a TCP stream with length prefixes or a datagram protocol with message boundaries).

ClientHello Message:

```
struct {
uint8 protocol_version_major;
uint8 protocol_version_minor;
uint16 cipher_suites_supported;   // e.g., a bitmask or ID indicating profile support
opaque client_random[32];        // 256-bit random nonce
// Optional fields
uint8 have_ephemeral_key;        // 0 or 1
if (have_ephemeral_key == 1) {
uint16 ephemeral_key_length;
opaque client_ephemeral_key[ephemeral_key_length];  // e.g., a serialized phi-ns public key or other
opaque client_ephemeral_commitment[??];            // maybe a q^2_masked for the clients ephemeral q, if needed
}
} ClientHello;
```

This is an example. In many cases, cipher_suites_supported might just be a single profile ID if ASL is simple (for instance, 0 = base, 1 = withDoors, 2 = PK-p mode only, etc.). The client_random ensures the handshake has entropy from client side as well.

ServerHello (ServerParams) Message:

```
struct {
uint8 protocol_version_major;
uint8 protocol_version_minor;
uint16 cipher_suite_chosen;
opaque server_random[32];
uint8 key_mode;     // 0 = PK-q (server public = q), 1 = PK-p (server public = p)
opaque server_public_key<1..+>; // If key_mode=0, this contains q (big-endian integer). If key_mode=1, contains p.
opaque q2_masked_value<...>;       // A fixed-length encoding of q^2_masked. For example, same length as M or other standard (must be defined by profile).
uint16 modulus_length;
opaque modulus_M[modulus_length];
opaque generator_g[modulus_length];  // same length as modulus to hold value
opaque initial_u[modulus_length];
// if key_mode = 1 (PK-p), include auxiliary data for q hiding:
if (key_mode == 1) {
opaque q_anchor[modulus_length];   // possibly q mod M or some
```

representative
uint16 star_count;
opaque star_projections[star_count][modulus_length]; // array of
star_count elements, each modulus_length bytes
// (star_count could be implicitly determined from length too)
}
} ServerHello;
Note: We treat p, q, M, g, u as big-endian byte arrays with a known length (modulus_length could be, e.g., 32 bytes for a 256-bit prime M). The star_projections array is an example of optional data; its content and use must be specified by the profile (the client needs to know what to do with them, which might be documented in an application profile spec rather than the core).

ClientEncryptedSecret Message:
struct {
opaque encrypted_secret_value<1..+>; // The ciphertext X (or C) from encrypting the clients secret.
// Optionally:
opaque handshake_verify_tag[32];   // e.g., a 256-bit tag for handshake verification (if implemented).
} ClientEncryptedSecret;
The encrypted_secret_value length should equal the length of modulus M (as ciphertext is an element of Z_M). If the secret is slightly bigger than M bits, multiple such structures could be sent or it could be split, but typically we'll choose M to be big enough.

ServerFinish Message:
struct {
opaque next_u[modulus_length];
opaque next_a[modulus_length];
opaque verification_ciphertext[modulus_length];
} ServerFinish;
In this, next_u and next_a are state for round 1 (if not already known; if we assume handshake round was 0, then round 1 is first data round possibly carrying this verification). The verification_ciphertext is the encryption of a known value (like all zeroes or some hash of transcript) using round 1's state. The client will decrypt it and check. If separate sequences are used, then perhaps the server's finish would use server's sending state sequence's first value. But let's not digress; the format stands.

After the handshake:
Data Message Format (general for any subsequent round):
struct {
uint32 round_number;
opaque u_value[modulus_length];
// a_value might be omitted if receiver can compute it:

```
// but including for completeness
opaque a_value[modulus_length];
uint16 ciphertext_length;
opaque ciphertext[ciphertext_length];
} ASLRecord;
```

Here, ‘ciphertext_length‘ might equal ‘modulus_length‘ if a single field, or could be a multiple if the plaintext was chunked into multiple modulus-size blocks (like if plaintext is very large, one might do multiple phi-ns-x transforms sequentially or in parallelthough more likely you’d just send multiple records).

If we treat each message to be max one modulus worth of data, ciphertext_length can be fixed to modulus_length except possibly the last block of a file transfer (but since phi-ns-x isn’t a streaming cipher, better to break each application message separately).

The ‘round_number‘ is 32-bit allowing for a large number of messages. In some contexts (like blockchain), the round might correspond to a block or tx number.

All multi-byte fields are in network byte order (big-endian) by convention.

For encoding integers (like p, q, M, u, a):
- They should be in big-endian byte arrays of a fixed length (preferably the length of M for consistency, padded with leading zeros if needed).
- The prime p or q in server_public_key can be shorter than M (e.g., if p is 160 bits and M is 256 bits). In that case it can be left-padded with zeros to some fixed length or encoded with a length prefix. Above we allow it to be a variable length field with a prefix for length (‘server_public_key<1..+>‘ means a variable-length vector as per TLS syntax).

Notably, these structures closely mirror TLS handshake messages albeit simpler since there’s no certificate chain, no separate signature (the encrypted secret and q^2_masked serve that purpose).

The protocol’s normative requirements:
- The client and server MUST verify each other’s values as described (e.g., q^2_masked and handshake decrypt).
- If any verification fails or an unexpected message is received, the implementation MUST abort the connection.
- All numeric fields are unsigned and non-negative.
- The primes p and q in the key must satisfy all conditions from section 3 (e.g., T composite, etc.) otherwise the protocol’s security assumptions break.

Finally, it’s possible to compress some messages in an optimized

implementation:
- For example, if the cipher suite is fixed and known, ClientHello could be omitted and the server could start first (like in some PSK modes). But the full handshake as described is the default.

With the message formats defined, an implementation can marshal/unmarshal these structures and perform the cryptographic operations accordingly. This textual specification should be sufficient to guide interoperable implementations.

5. Example Use Cases

In this section, we discuss how ASL can be applied in various scenarios, highlighting the benefits and any special considerations for each.

5.1. HTTPS Without TLS (Web Security)

ASL can be used as a drop-in replacement for TLS to secure HTTP connections (hence "HTTPS without TLS"). In this use case, a web server would have an ASL server key (Phi-Ns based) instead of an X.509 certificate with an RSA/ECDSA key. The browser (client) would either have the server’s public key (p or q and q^2_masked) pre-loaded (through some out-of-band means like DNS records, .well-known endpoints, or built into the application) or would use a trust-on-first-use approach (similar to how SSH handles host keys).

Establishment: When the browser connects to the server, it performs the ASL handshake:
- The server proves its identity by its ability to decrypt the ClientEncryptedSecret and by having a q^2_masked matching a published value. No certificate chain is needed; the trust is directly in the server’s public key. This simplifies the handshake (fewer round trips than a full TLS handshake with certificate verification) and avoids reliance on certificate authorities.
- The handshake, as designed, can be as low as 2 round trips (ClientHello -> ServerHello -> ClientSecret -> ServerFinish), similar to TLS 1.2 with no client auth. With some optimization (or early data concept), it could even be reduced, but let’s assume 2 RT for fairness with TLS 1.3 which uses 1 RT but needs certificate in that RT.

During Data Transfer: The HTTP requests and responses are sent encrypted as ASL records:
- The client would send a GET request as plaintext into an ASLRecord payload. The server decrypts it, processes it, and sends back the response in one or more ASLRecords.
- The stateful nature of ASL means each request/response increment the state. In HTTP (which is mostly request-response sequential on a given connection, at least in HTTP/1.1 or HTTP/2 streams), this aligns well.

Each HTTP message corresponds to one or more ASL rounds. If pipelining or multiplexing (like HTTP/2), separate logical streams could either use separate ASL sequences or be serialized within one sequence - a multiplexer would handle ordering. Simpler approach: one ASL session per HTTP/2 stream to avoid head-of-line blocking; but that might be overkill. It’s easier to treat the whole connection as one sequence and trust ordering at transport.

Benefits:
- No reliance on specific cryptosystems that might be broken by quantum computers (RSA/ECDH). ASL is built on a new assumption believed to be quantum-resistant.
- Compact public keys: A Phi-Ns key with, say, 192-bit p and q plus structure can be far smaller than an RSA 2048-bit modulus. It could be comparable in size to ECC keys, but with PQ security.
- The handshake does not transmit certificates, meaning less data transferred (helpful for IoT or constrained networks), and avoids the need for verifying certificate chains (saving computational cost on clients).
- Privacy: Because no certificates are sent, there’s less risk of an eavesdropper collecting certificate info to track servers. The server’s public key might be well-known via other means, but at least it’s not transmitting a whole identity chain every time.
- Implicit authentication: If the public key is obtained through DNSSEC or some secure channel, the user agent can authenticate the site similarly to how it would with a certificate.

Considerations:
- Bootstrapping trust: Without the CA system, how does the client know the server’s key is legitimate? This could be solved by a one-time CA-like attestation (maybe the domain publishes its ASL key in a certificate-like format signed by a CA, or uses DANE via DNSSEC). This is a deployment rather than protocol issue.
- Performance: The encryption and decryption operations (involving modular arithmetic and possibly searching through quadratic solutions) must be efficient. For web traffic, throughput is important. Profile M or L from Phi-Ns is appropriate (128-256 bit primes, recursion if needed). Those should be efficient on modern CPUs. Preliminary estimates show Phi-Ns operations can be done in microseconds, comparable to RSA or DH in many cases, but this should be validated.
- Session resumption: As with TLS, one might want to resume a session without full handshake (to avoid repeating heavy ops). ASL could implement resumption by having the server issue a session ticket (encrypted blob containing the shared secrets or a reference) that the client can present next time. Since ASL is symmetric once established, resumption is straightforward: it’s essentially storing step_u and current state etc. Implementation of session resumption is an added layer; the base protocol doesnt preclude it.

Overall, ASL could provide a secure channel for HTTPS traffic with potentially simpler mechanics and post-quantum security built-in. The normative behavior in this use case is exactly as described by the protocol; the main difference is in how keys are distributed and managed.

5.2. Blockchain Transactions

In some blockchain systems (particularly permissioned or consortium chains), there is a need to encrypt transaction payloads on-chain such that only authorized participants can read them, while others can still validate some aspects (like knowing it was a valid encryption under a known key and state).

ASL can be adapted to this scenario, as was partially demonstrated in the "Rapport" simulation:
- The blockchain network can have a static public key associated with the chain or with each participant. For example, each participant (node) might have an ASL key pair. If a transaction is intended for a specific group, those members share an ASL channel (with shared secret).
- The "Genesis block" or chain configuration would publish relevant parameters: M, g, and perhaps a common key if one key pair is used system-wide for encryption (or each participant’s public keys if multiple).
- In the simulation, they had: ‘Genesis publishes M, g, p_public, q_anchor, q2_masked + star projections‘. This is essentially the network announcing, "here’s the public key (p_public) and associated data you’ll need to encrypt transactions to us; trust is established by q2_masked etc."
- Each transaction included ‘(u_k, a_k, C_final)‘:
  * ‘u_k, a_k‘ correspond to the public state of that round of encryption.
  * ‘C_final‘ is the ciphertext (potentially using an Eq6-like projection as shown: ‘C_final = (X + q)^2 - T * a_k^2‘ in their logs).
- Miners/validators who are not privy to the secret cannot decrypt the payload, but they can verify that the transaction is well-formed in terms of state linkage:
  * They can check that the ‘u_k, a_k‘ values are correctly derived (a_k = (u_k + inv(u_k))/2 mod M, which anyone can do).
  * They can check that ‘u_k, a_k‘ follow the previous state’s progression if the chain enforces an order (the logs show each tx had a round number and they likely ensured u_{k} was derived from u_{k-1} by the shared step).
  * They can store the new state (so that the next transaction from that channel can be verified against it).
  * If ‘C_final‘ uses a construction like Eq6, perhaps the validators can verify that for the given ‘C_final‘ and known public values (q^2_masked might help here), there’s a consistency: actually, without

knowing T, they can’t directly verify (X + q)^2 piece. But if they trust that any invalid ciphertext would not be producible without secrets, they might just assume a valid-looking state means it’s fine. Alternatively, they might require the sender to later reveal plaintext and then verify by re-encrypting (commit-reveal scheme, which was mentioned in the notes).
- The participants who hold the secret (the ones authorized to decrypt) will retrieve the transaction, use their private key to decrypt and read the message. They can later reveal the plaintext if needed for dispute resolution, and anyone can recompute ‘C_final‘ to confirm it matches (as shown by the log lines: they did a REVEAL verify by recomputing C_final from plaintext and it matched).

ASL fits this because:
- It provides a way to have evolving state (so you don’t reuse the same key for every transaction, which is good for security and unlinkability).
- Only those with secret (p, q, etc.) can create valid ciphertexts, which means unauthorized parties cannot forge a transaction that would be accepted into the chain (unless they somehow guess the correct state and produce a valid encryption by chance, which is astronomically unlikely). This gives implicit access control at the encryption level.
- If a participant leaves or is to be excluded, the key can be rotated (generate new p, q or just re-randomize abR to effectively create a new key that yields a new q2_masked, etc.). Future transactions would then use the new key, and old ones remain encrypted under the old key which the leaving participant may still have, but new data they can’t access. This is more of an operational procedure.

For a public blockchain scenario where anyone can be a reader, ASL isn’t needed because data is public. But for consortium where privacy is needed, ASL could run on top of the blockchain as an application layer or integrated with consensus so that only intended parties see content.

One must also consider performance: blockchain blocks may contain many transactions, so encryption and decryption should be efficient. ASLs small key sizes and avoidance of heavy operations per message (just modular multiplications and one inversion per layer) are beneficial. Also, the on-chain footprint of each encrypted tx is just a few numbers (u, a, C) which are all reasonably small (the size of M, e.g., 32 bytes each for 256-bit security). That’s much smaller than storing, say, an RSA encrypted blob of symmetric key plus the cipher.

The demonstration logs correspond to an ASL channel where multiple parties (Alice, Bob, Carol) were sharing state (they all had the keys to update u and could each encrypt messages). That shows ASL supports multi-sender scenarios if they coordinate state (like in a group, you need to ensure only one uses a given round or they need to coordinate

incrementing state). Perhaps in that demo they all share step and doors, and they just pick next round sequentially when they send a message.

In implementation, to avoid collisions, a group might require a leader to assign round numbers or have each include the round in transaction (which they did) and miners ensure no duplicates and order by that.

Benefits:
- **Confidentiality**: Sensitive transaction data (like financial details, private messages, etc. in a block) remain confidential to authorized parties. The chain still provides integrity and order.
- **Authenticity**: Only someone with the groups shared secret can produce a valid ciphertext for the next state, preventing outsiders from injecting transactions that would later decrypt to something. It acts like a group signature in some way (only group members can "sign" valid ciphertexts).
- **Post-quantum group security**: Many current blockchain privacy techniques use ECC (for addresses or for encryption in certain layer-2 solutions). ASL offers a different hardness assumption which might be safer if ECC is compromised by quantum computers.

Considerations:
- Key distribution: Initially, how do group members get the secret key? That’s outside ASL’s scope (could be done in person, or using another secure channel).
- State synchronization: In a distributed setting, ensuring everyone has the same current state is important. The blockchain itself helps because the latest block contains the last transaction and thus the last state (so even if a node was offline, by reading the chain they can catch up on u_k).
- If the chain allows parallel channels (like multiple disjoint sets of participants each with their own state), each such channel would have its own ASL context identified by something like a channel ID, and state tracked per channel. This is achievable as well.

5.3. Implicit Client Identity Authentication

Traditional protocols often authenticate a client by an explicit credential (username/password, token, or client certificate). With ASL, especially in a closed ecosystem (corporate network, IoT deployment), a client’s identity can be tied to possession of a secret key, and the handshake itself reveals which key (and thus which identity) is being used, without needing a separate authentication step.

For example, consider an enterprise where each device or user is issued a unique ASL key pair (or at least a unique secret derived from a master key). The server (or gateway) maintains a list of authorized public keys (or their commitments). When a client initiates ASL:

- The client might include an identifier of which key it’s using (or this could be implicit if the server can try each known public key to see which matches the q^2_masked provided, but it’s easier if client sends an ID).
- The server receives the handshake with, say, clients p_public and q^2_masked. If it finds a match in its database, it knows which client this is. It then proceeds to complete handshake and now all messages are encrypted.
- If an attacker attempts to impersonate a client, they won’t have the client’s secret key, and thus cannot complete the handshake properly (they’d fail to decrypt or fail to prove knowledge of q by producing correct q^2_masked).
- Thus, possession of the key is the identity proof. No password or token needed at session time.

This "implicit identity" is analogous to how SSH key-based auth works: if you can produce the correct signature with your private key, you are that user. Here, if you can produce a valid handshake with the key, you are the authorized device.

Another angle: Suppose the servers ASL key is public for all, but the shared secret derived in handshake is based on something the client knows. Actually, that is more like password-authenticated key exchange, which ASL by itself isn’t doing. However, we could incorporate a client secret in handshake by having the clients encrypted message depend on a password somehow (e.g., plaintext includes a password that the server knows and can verify after decrypting K_s). But that is a custom protocol on top of ASL handshake, not inherent.

Better to just give each client a key:
- It could be a full Phi-Ns key pair or perhaps just a symmetric key pre-shared to derive something for handshake. For instance, the client and server could share a long-term secret and use that to generate the handshake’s K_s (like a PSK mode). The client could encrypt something that includes evidence of the PSK. However, describing that fully is beyond scope; suffice to say ASL could be extended for PSK (pre-shared key) mode by mixing a known secret into K_s derivation or verifying a tag.

In pure public-key mode:
- The server might not want to store all clients’ private keys obviously, it just stores their public half and maybe q^2_masked to verify integrity.
- Each client effectively has a "certificate" by virtue of their public key being recognized. The difference is they don’t need a separate signing step; the act of handshake doubles as the signing because only the real client could produce that encrypted pre-master secret that results in a meaningful shared key.

Use cases:
- **IoT**: Each device has an ASL key burned in. The central server uses ASL to communicate, and it immediately knows which device it is talking to by the key. If a device is compromised, the admin can remove its public key from allowed list (revocation).
- **VPN**: Users could authenticate to a VPN server by ASL key (instead of username/password). The connection establishment (like IKE or TLS in OpenVPN) could be replaced by ASL handshake using the user's key. If successful, the server associates that session with the user identity of that key. This is certificate-like but without X.509 overhead.
- **Client certificates alternative**: In web, mutual TLS is rarely used because it's cumbersome. With ASL, if a client has a key, they can prove identity seamlessly as part of the handshake without complicated certificate UI. Of course, the client still needs to have the key pair available (maybe in OS keystore).

Security:
- The strength is that the client key is as hard to forge as breaking Phi-Ns. There's no online dictionary attack vector since there's no password, it's public-key crypto.
- The server must protect its list of allowed keys and any mapping to user accounts.

Privacy:
- If clients present a persistent public key, that can be used to track them across sessions. In some contexts that's fine (enterprise), in others maybe not (web). There are ways to mitigate tracking by rotating keys or having the server assign ephemeral identities. But that's deeper into privacy design.
- The handshake itself reveals which key (hence which client) is connecting (because p or q^2_masked is essentially an identifier). If anonymity is needed (like in some protocols), one might not want that. However, this is similar to client certificates anyway.

5.4. Secure Messaging

Consider an end-to-end encrypted messaging application (like a chat app). Typically, such apps use protocols like Signal, which involve Diffie-Hellman key exchanges, X3DH, double ratchets, etc., to ensure forward secrecy and authentication. ASL could serve as an alternative core for some of those steps, or even the entire session.

Scenario:
- Alice wants to send an encrypted message to Bob. She knows Bob's ASL public key (obtained from Bob's profile or directory).
- They could perform an ASL handshake to establish a shared secret and then send messages. But if they are not online at the same time for a handshake, you could do an offline setup:

* Possibly one party can precompute a handshake message and the other completes it when they come online. This is tricky because ASL as described is interactive. But maybe Alice can send "ClientHello + ClientEncryptedSecret" to Bob (through the server) while Bob is offline. Bob when coming online can process it and send "ServerHello + ServerFinish"? The ordering would differ because normally server goes second. Alternatively, they'd just wait until both are online.
- More straightforward, if asynchronous: treat each message as its own ASL "handshake+data" maybe, or use pre-shared static keys with ratchets on top.
- However, if interactivity is allowed (both online concurrently, like voice call or live chat session), they can do a full ASL handshake directly peer-to-peer and then chat with messages as ASL records.

In such a chat:
- Both peers likely have devices with their key pairs (like each user has a long-term key).
- Mutual authentication can occur by each verifying the other's q^2_masked through a fingerprint exchange (like scanning a QR code of the key fingerprint out-of-band, similar to how Signal verifies identities).
- Once they have a session, each message is encrypted with evolving state. This inherently provides **forward secrecy** across messages: an attacker who compromises a key after some messages were sent cannot decrypt the earlier messages if those messages involved state that isn't reproducible from just the static key. Actually, careful: if an attacker gets p and abR, can they decrypt old messages? They still need the per-session shared secret K_s or at least the states. If those states were derived from K_s which was ephemeral and not stored, then yes, forward secrecy holds. So to ensure FS, one or both must contribute an ephemeral in handshake (like ephemeral Diffie-Hellman in TLS). ASL handshake as given uses the static server key and an ephemeral client secret K_s. But the static server key by itself means if server key is compromised, the handshake could be decrypted because the attacker can simulate it? Actually, they'd need the transcript. If someone logs all handshake messages and later the server key gets stolen, could they derive the session keys? Possibly yes, because with p, q they could decrypt ClientEncryptedSecret (X_secret). So without an ephemeral server key, a long-term key compromise does compromise past sessions. So strictly, the current ASL handshake is not forward-secret if the server's (or main key holder's) static key is compromised. Its like RSA TLS: break RSA key, you can decrypt old pre-master secrets.
- So for messaging, they likely want forward secrecy. That can be achieved by having both parties use ephemeral keys in an ASL handshake (like I mentioned, each could generate an ephemeral phi-ns pair, exchange them in the handshake and derive secrets combined). Or simpler, just do a fresh handshake with ephemeral keys for each message or each session, but that might be too slow if frequent.

- Alternatively, ASL could be integrated into a double ratchet scheme: for each message, the state update acts like a ratchet step since it changes the keys. Provided that the state updates are not reversible without the prior key, then even if one message key is compromised, previous keys are safe. But if the static key is compromised, all might be at risk.
- Possibly combine static identity keys (for authentication) and ephemeral session keys (for encryption). This is akin to how Axolotl/Signal does it: you have identity keys and ephemeral one-time keys. One could design an "X3ASL" handshake: exchange static keys signed, plus ephemeral keys in an ASL style.

The details can get complex. The main point: ASL provides a structure that can do the heavy lifting of encryption with evolving keys, but one might still need to incorporate ephemeral keys for full perfect forward secrecy if desired.

If forward secrecy is less a concern (say within a short live chat session), and key compromise is unlikely in real-time, ASL as is may be fine.

Benefits for messaging:
- Small messages overhead: each message only needs to carry a small state and ciphertext. Compare to, e.g., sending a whole new RSA-encrypted symmetric key per message (which some naive systems do), ASL is more efficient.
- No need for additional MAC each message if relying on state.
- The same library that does handshake sets up continuous encryption, simpler architecture.

Group chats:
- If multiple parties in a group chat, they can share one ASL channel (like the blockchain example, but now interactive).
- Alternatively, pairwise channels or using group keys (beyond our scope).
- ASL’s design with hidden factors could be exploited in creative ways for group: e.g., each group member could hold a piece of R such that only if they cooperate can they decrypt? That’s speculative and not standard ASL usage.

5.5. Encrypted Tunneling (VPN)

A VPN or any encrypted tunnel (like an IPsec replacement, or a simple point-to-point tunnel between networks) can use ASL to establish a secure link.

Typically, IPsec has IKE for key exchange then uses symmetric ciphers for bulk data. TLS-based VPNs (like OpenVPN) do similar with TLS

handshake then symmetric cipher (AES) for data.

With ASL, we can unify that: the handshake and the data encryption use the same primitives and keys:
- The gateway and client perform an ASL handshake to authenticate (the gateway proves identity via q^2_masked, client perhaps via PSK or certificate equivalent).
- They derive a shared secret and trajectory parameters.
- Then they start forwarding IP packets or frames through the ASL channel. Each packet is encapsulated in an ASLRecord.
- Because of stateful encryption, this looks similar to a stream cipher in usage: packets go in order. If out-of-order delivery is needed (like multiple parallel packet flows), one might consider independent channels for each or accept the sequential requirement (which might add latency if one packet is lost, subsequent can’t be decrypted correctly until maybe that state is recovered or skipped).
- Perhaps treat each packet independently by resetting state if you don’t want head-of-line issues. But that loses the benefit of evolving state (though you could still do a quick re-handshake or treat each as separate message with small state changes).

For a reliable transport or for link encryption where ordering is ensured, ASL is fine.

Post-quantum benefit:
- Many current VPNs would need to switch to PQ algorithms; ASL is a candidate with unique properties.
- As a pure arithmetic algorithm, it might be easier to implement in embedded routers without special hardware (and not be subject to certain side-channel issues if careful, though side-channel protection would be needed as always).
- The overhead per packet is just a few bytes plus the computational overhead which should be manageable (maybe on the order of a few hundred multiplications per packet, which on hardware can be optimized).

Integration:
- ASL could be a new protocol at either layer 4 (like a custom reliable protocol carrying encrypted data) or layer 2/3 (like how IPsec ESP works).
- One could even embed ASL within existing protocols by using reserved bits (for example, encode u and a in an IP option header - not realistic publicly, but in a closed system who knows).

Security:
- MITM is prevented because the handshake authenticates the server (and possibly client). Attackers on the path cannot decrypt or inject because they can’t predict the state or produce valid ciphertext.
- Even if one packet is captured, without the key they can’t glean

anything. And if one tries to replay a packet, the state will have moved on so the receiver would find the round number out of date and drop it, thus providing replay protection.

Comparison:
- Compared to IPsec’s use of AES-GCM (which provides confidentiality and integrity with an IV and counter per packet), ASL provides a somewhat analogous function: the evolving state is like a counter/IV, but with secret evolution making it unpredictable to outsiders (whereas in AES-GCM the counter is usually not secret but just a nonce).
- ASL inherently authenticates the data because forging requires the key. AES-GCM similarly prevents forgery due to the tag. ASL’s approach might lack an explicit tag, but the non-linearity could serve similarly. Still, one might consider adding an explicit lightweight tag if paranoid.
- Throughput: AES is faster with hardware support. ASL is software-based, but if its operations can be vectorized or offloaded (like modular multiplication can be done with big integer libraries, maybe not as fast as AES-NI instructions though). For moderate speeds (e.g., <1 Gbps) ASL should be fine in software on modern CPUs. At very high speeds, one might explore hardware acceleration (maybe FPGAs could accelerate mod M operations, or a custom ASL co-processor).
- The benefit is not needing separate systems for key exchange and encryption - it’s unified.

6.
4.x Encryptor Operation and Knowledge Model

This section specifies precisely the role, capabilities, and limitations of the encrypting party in the Authenticated Secured Layer protocol using
Phi-NS.

The encryptor does NOT possess the private root p, the value q, nor the quadratic gap T = q^2 p^2. The encryptor is not required to know any secret factorization, atom ordering, or private trajectory parameters.

The encryptor operates exclusively on public, evolving state values.

For each encryption round k, the encryptor is provided with a public state value u_k Z_M*.

From this public state, the phase value a_k is deterministically derived as:

    a_k = (u_k + u_k^{-1}) inv2 mod M

where inv2 = 2^{-1} mod M.

The encryptor then applies the sequential Phi-NS transformation to the message.

Let the plaintext message be represented as an integer m Z_M.

The encryption recurrence is defined as follows:

```
p_0 = m

For i = 1 ... L:

    y_i = (p_{i1} + a_k) mod M

    p_i =
        ((y_i^2  y_i + p_{i1}  a_k)  y_i^{-1}) mod M
```

where y_i^{-1} denotes the modular inverse of y_i in Z_M.

The ciphertext output of the encryption process is defined as:

```
C = p_L
```

Optionally, the ciphertext MAY be further bound to the quadratic structure by applying a quadratic projection, as defined elsewhere in this document.

At no point does the encryptor require knowledge of p, q, T, or any private atom or star structure. The encryption process is entirely public and deterministic with respect to the public state u_k.

The security of the scheme relies on the infeasibility of reconstructing the inverse trajectory without possession of the private parameters and the correct internal ordering of transformations.

Security Considerations

This section analyzes various security aspects of ASL, reiterating some points from earlier sections in a concise manner, and adding any additional considerations. ASL is designed to provide confidentiality, integrity, and authenticity in the presence of active adversaries, and to resist both classical and quantum cryptanalytic attacks. However, as with any cryptographic protocol, correct implementation and parameter choices are critical.

6.1. Man-in-the-Middle (MITM)

ASL explicitly addresses MITM attacks through its handshake design. An

MITM is an attacker who intercepts communications and attempts to impersonate each party to the other. In the ASL handshake:
- The server authentication is based on possession of the secret prime (or its factors). The client verifies the server by checking the ‘q^2_masked‘ value against a trusted reference and by ensuring the server can decrypt the client’s secret message. An attacker in the middle who does not have the server’s private key cannot satisfy both conditions. They might forward messages to the real server (attempting a relay attack), but the handshake verification (e.g., the final encrypted verify message with state binding) will fail unless the MITM lets the actual server complete the handshake, in which case the MITM isn’t actually decrypting anything (just tunneling).
- If the MITM tries to use their own key pair to talk to the client, the ‘q^2_masked‘ check will fail (client won’t recognize this rogue server’s identity as legitimate). If they try to use their key with the real server, the server will similarly not recognize the client if client authentication is expected, or the server will complete handshake but then the MITM can’t use the session because the client side didn’t complete it with them properly.
- Thus, as long as the client has a correct trusted copy of the server’s public-key identifier (and the server is not compromised), a MITM cannot pretend to be the server. This is equivalent to saying ASL, like TLS, assumes an authenticated server public key distribution. If that initial distribution is via a secure channel (like a known fingerprint, or DANE, or a CA-signed object), MITM is thwarted. If it’s not (like no verification, just take whatever comes first), then a MITM during first contact could trick the client into trusting a fake key (TOFU scenario). That is not a protocol flaw but a bootstrapping issue.
- For client authentication (if used), a MITM similarly cannot masquerade as the legitimate client to the server because they cannot produce a valid handshake under the client’s public key if they don’t have the private components. The server will either not accept the unknown key or, if the MITM tries a known client’s ID but doesn’t have key, it will fail at handshake verification.

ASL also binds handshake and record layer tightly via the evolving state. After handshake, a MITM who didn’t fully participate (and succeed in impersonation, which we argue they can’t if checks are done) cannot inject data later:
- Suppose a MITM recorded some ciphertext and tries to replay or inject it. If they don’t know the current correct state or key, the receiver will find the ‘u_k‘ is not the expected next state (replay detection) or will fail to decrypt properly. Because the state transition is one-way and secret-driven, the MITM cannot guess a future valid state to inject fake ciphertext.
- Meet-in-the-middle attacks in key search (where attacker tries to split a key into two parts and compromise one part at a time) do not apply to ASL’s core because the key isn’t used in a way that halves the

problem. The entire structure has to be guessed as a whole (see section 3.4 on no partial disclosure).
- A specific MITM concern is protocol downgrade: if there were multiple cipher suites or versions, an attacker could try to force the use of a weaker one by dropping messages. ASL's ClientHello/ServerHello version negotiation should be done with care (like TLS, the client should abort if server picks something not actually offered). We assume any alternate modes (like turning off injection or using smaller keys) are either not present or if present, chosen in a way that a MITM cannot cause insecurity (by always using secure defaults or requiring client consent).

In summary, if ASL is executed correctly with authenticated keys, MITM is effectively mitigated. Implementers MUST ensure to perform the verification of 'q^2_masked' and handshake results. Skipping those checks or ignoring authentication (for convenience or performance) would open a door to MITM, just as not checking a TLS certificate would.

6.2. Collision Resistance

"Collisions" in this context can mean a few things:
- Two different keys yielding the same public parameters (commit collisions).
- Two different plaintexts yielding the same ciphertext under different keys or states (encryption collisions).
- Collisions in the sense of cryptographic hash (if we used any).

Key/Public Parameter Collisions:
- Could there be two distinct secret tuples (p, a, b, R) and (p', a', b', R') that produce the same public outputs (either same q or same commit)?
- Having the same q would mean either p = p' or a scenario like p and p' both associated with the same q (which from q^2 - p^2 = q^2 - p'^2 implies p = p', so either p = p' or p = -p' mod something but since these are integers, basically p equal or just sign difference irrelevant here because primes are positive).
- So no two different p values should yield the same q.
- Could two different abR structures yield the same serialized commit? The design of serialization and commitment (assuming an ideal hash or a well-chosen masked scheme) should make collisions computationally infeasible. We rely on that as a design assumption: e.g., if q^2_masked is chosen as say q^2 mod N for some number N, an attacker might try to find a different q' that yields same q^2 mod N, but that would require solving q^2 q'^2 (mod N), which often has the trivial solutions q' q (mod N). If N is large (like a 512-bit number) and not specially structured, the chance of a random different q colliding is negligible. If an attacker found a colliding key that produces the same masked commit as a target, they could impersonate that target's identity. To

guard against that, one should choose the commit function to be collision-resistant (like a cryptographic hash or sufficiently large modulus to make solving that congruence as hard as guessing q).
- We consider that aspect in design: it’s RECOMMENDED to use a proven collision-resistant method for the commitment. Using a hash like SHA-256 of the serialized abR would be safest (practically no collisions). Our avoidance of SHA in core was philosophical, but one might reintroduce a hash just for identity safety. If not, the masked approach used must be analyzed for collision risk.
- In practice, the chance of two legitimate users picking keys that collide in commit is astronomically low if done right. The bigger risk is an attacker deliberately computing a colliding key pair if any weakness is found in commit. That’s a niche attack; presumably as hard as breaking the scheme itself if commit is well-designed.

Ciphertext Collisions:
- Could two different plaintexts produce the same ciphertext under the same key and state? Ideally not, as that would violate invertibility for the legitimate decryptor (they wouldn’t know which plaintext was intended). Given phi-ns-x is a permutation on Z_M for fixed a_k (except the small chance of hitting the y_i = 0 case), it should be one-to-one (maybe not strictly a permutation on the entire space if y=0 is excluded, but on the domain of allowed inputs it should be injective).
- The transform looks invertible if you have the secret; it’s likely that for each possible X there is exactly one plaintext that would produce it (or at most a small constant number given the quadratic nature, but the decryptor can figure out the correct one via key).
- So collisions in this sense are not expected; each encryption mapping is unique per message. There’s no known smaller subspace that two different messages would converge to the same X because that would imply solving p_L = constant for two different p_0 given the parameters a difficult equation with no reason to have two solutions except trivial symmetric cases, which the key structure would break.
- If optional masking like Eq6 is used, that formula is still injective given secret info; but an attacker might see multiple ciphertexts that numerically collide (e.g., if a certain pattern made two different X yield same C, they’d be indistinguishable to attacker, but decryptor would not confuse them because they’d decrypt differently). However, one must ensure that doesn’t lead to a chosen-ciphertext vulnerability (shouldn’t, because attacker can’t control X without key anyway).
- Also, sequence numbers (round) effectively make even identical plaintext encrypted at different times yield different ciphertext because a_k (hence the transform) changes each round. This provides semantic security (no pattern leak if same message repeated, unlike a deterministic cipher).

Hash Collisions:
- If any hashes (like for commit or handshake verify) are used,

collision resistance of those is required to avoid impersonation or tampering. Standard hashes like SHA-256 are fine. If an attacker found a hash collision in, say, the commit function, they could potentially produce a different abR that yields same commit, undermining key authenticity. This is theoretical (depending on commit scheme).
- Or if using hash for handshake transcript, collisions are not a big worry unless an attacker could craft two different transcripts that produce same hash and trick one party, but that seems remote and anyway if handshake was manipulated that far, likely other parts would break.

Overall, collisions are not seen as a practical threat in ASL when parameters and algorithms are chosen appropriately. Nevertheless:
- **MUST**: The commit function for key identity must be collision-resistant (to at least 128-bit security). Using a secure hash function is recommended unless a purely arithmetic method with equivalent hardness is in place.
- **MUST**: The encryption function should never be intentionally used in a way that non-injective behavior could occur. For instance, avoid using an M that is too small such that the plaintext space is much larger than M (which could cause wrap-around collisions).
- **MUST**: Each encryption round use a unique state to avoid trivial collisions (like encrypting two messages with same state would produce same ciphertext if plaintexts equal; but state reuse must be prevented like IV reuse in any cipher).

6.3. Secret Leakage and Oracle-Free Design

One of the core security principles of Phi-Ns and ASL is the absence of any "oracle" that leaks information about the secret key or intermediate states:
- During handshake, the server does not disclose anything that correlates directly with its secret. The ‘q^2_masked‘ value is a one-way function of q. The encryption/decryption process either succeeds or fails but does not reveal partial info: if an attacker guesses a wrong p and tries to decrypt something, they either get a wrong result or no result, with no indication of how far off they were.
- The protocol does not use branch conditions or different responses that depend on secret values (at the spec level). Implementations should ensure to follow this: e.g., a naive implementation might accidentally take a different amount of time to decrypt based on how factors align, or return distinct error messages for different failure reasons. That could leak info via side-channels or behavior. An implementation **SHOULD** use constant-time arithmetic where feasible and treat all failures uniformly (just "handshake failed" without saying why).
- In the data phase, an attacker cannot query the decryptor with arbitrary ciphertexts to get hints. If they send a malformed ciphertext, the receiver will drop it or potentially respond with a generic error or de-sync. There is no decrypt-oracle giving success/failure on a per-

packet basis in a useful way; at best, an attacker could flood with random ciphertexts and see if the receiver at some point processes one (meaning it was valid). But the odds of guessing a valid ciphertext and state is essentially zero. The receiver might also have anti-replay and anti-desynchronization logic (like not resetting state on a bad message, etc.) to avoid giving even timing clues. In short, forging a valid ciphertext is as hard as breaking the crypto, so no oracle helps.
- The public values u_k and a_k leak no secret info by design. ‘a_k‘ is related to ‘u_k‘ by a simple formula with no secret. ‘u_k‘ evolves by multiplication with g^step_u. Now, one might worry: if an attacker sees two successive u values, can they solve for step_u? This is essentially solving g^step_u = u_{k+1} * inv(u_k) mod M. That’s a discrete log problem. If M is large enough (e.g., 2048-bit prime), discrete log is infeasible classically and also by known sub-exponential algorithms. But note: discrete log is not NP-hard in general, it’s sub-exponential (like 128-bit security might require 3072-bit modulus). But also note: a quantum attacker could solve discrete log mod M in polynomial time (Shor’s algorithm) if they had a quantum computer.
- Does that mean if someone had a quantum computer, could they find step_u and then predict future states? Yes, if M was a standard prime field, Shor would find step_u from a single state pair. That would let them predict all future u/a. But would that let them decrypt content? Not directly, because they still don’t have p or the door values. They could mimic the state sequence and attempt to encrypt their own messages or maybe mount some known plaintext attack if they guess content. However, just knowing state might help them attempt to brute force p by having more structured equations. But arguably, even with known u sequence, the core phi-ns problem remains (they have q possibly if server is PK-q, they have p if PK-p). Actually, if PK-q mode and attacker gets step from states, they know how u evolves. They could then generate u and a for future, but still can’t decrypt without p. They could however attempt to send messages (since encryptor only needs a and public anchor q). Uh oh: if attacker knows step_u and has q (which is public in PK-q), they could now encrypt messages that the server would accept because they can predict the state and they know q. That breaks authenticity in PK-q mode if step is exposed. This is why leaking step_u is dangerous. So we must rely on discrete log hardness for adversaries.
- Therefore, to avoid that, one might choose M to be a safe prime and g a generator, making discrete log as hard as possible for given size. For PQ safety, though, discrete log is vulnerable to quantum. But the idea is the attacker still couldn’t decrypt past messages, but they could inject future ones if they break step. This suggests an interesting point: in a post-quantum scenario, even if phi-ns core is safe from quantum (no known algorithm to recover p from q easily), the use of a standard finite field for state might be a weakness because of Shor’s algorithm. If a quantum adversary can derive step_u quickly, they can impersonate a client or server in sending data (though they’d still need to forge the proper door values if used).

- Potential mitigation: one could use a different group where discrete log is also thought PQ-hard (like a large non-abelian group or some lattice structure) for the state sequence. That becomes complex. Alternatively, ensure door values (stars) are used, because even if attacker predicts a and tries to encrypt, if they don’t know b_i sequence, their ciphertext will be wrong. So the secret injection saves the day: it means even with knowledge of step and q, attacker can’t create valid ciphertext because they’d miss the b_i.
- So it’s important: **If quantum adversary is a concern, use door injection or another secret element in encryption beyond just a and q**. This way, predicting the public state isn’t enough to forge messages.
- Also, after quantum, discrete log mod a large classical prime might become easy, so step would be known. However, recovering p from q (the phi-ns problem) might still be hard (super-exponential). So confidentiality might hold, but authenticity could suffer without the extra secret in each encryption. Therefore, for long-term PQ robustness, it’s RECOMMENDED to enable the layer injection feature with secret values (stars/doors) when using ASL if one expects adversaries with quantum capabilities that could target the state update.

Side-channel leaks:
- While not explicitly in protocol, implementers must watch for timing, cache, and other side-channels. For example, big integer operations may vary in time based on operands (leading zeros, etc.), which can sometimes leak bits of secrets. If a side-channel attacker on the same machine could observe decryption time variations, they might glean info about p or R. It’s a complex analysis for phi-ns, but general best practice: use constant-time arithmetic for key ops, avoid secret-dependent branches or memory access.
- Storing of secrets: The private key (p, a, b, R, and seeds) should be protected in memory. After use, if a session key K_s is derived, it should be zeroed out after use or when session ends. Similarly, if recursion adds additional secrets, treat them carefully.
- Randomness: The handshake relies on random ‘K_s‘. If the clients random generation were poor, an attacker might guess K_s and break that session. Also, the servers seed for key generation must be truly random, else the key structure might have hidden correlations. So a robust CSPRNG is needed for keys and nonces.
- The protocol has no dependence on any external random oracle at runtime besides those nonces and ephemeral secrets.

In summary, ASL’s design avoids deliberate information leakage. Implementers must maintain that property by following cryptographic coding standards. The protocol is safe against chosen-message attacks since an attacker cannot cause the legitimate parties to output something that reveals the key (there’s no encryption oracle giving out something like plaintext bits or error messages keyed on secret). The worst an attacker can do is send junk and get disconnected.

6.4. Post-Quantum Resilience

The cryptographic strength of ASL in a post-quantum setting is a key motivation for its design:
- The Phi-Ns problem (recovering p given q and commit) has no known quantum sub-exponential attack. Grover’s algorithm gives at best a sqrt speedup to brute force, which is negligible if the key space is super-exponential. The abR structure can be made large enough that even a quadratic reduction is insufficient.
- To contrast: RSA or ECC fall to Shor’s algorithm easily. Lattice-based schemes have better PQ standing but often larger keys and uncertain new problems. Phi-Ns offers another independent hardness assumption.
- That said, one must consider *all* parts of the protocol under quantum threat:
  * As mentioned, the discrete log in Z_M could be a weak link (since thats traditional math). If a sufficiently large quantum computer existed and M was standard size (like 2048 bits), an attacker could find step_u, step_star etc. If they also have q (in PK-q mode), they could then attempt to impersonate as discussed. If in PK-p mode (q secret), knowing step doesn’t directly give them ability to decrypt or send, since they still lack q and the injection secrets. So PK-p plus injection seems robust even against that quantum capability, because they’d still face the main phi-ns puzzle or have to solve for q.
  * If for some reason we used a known hash function for commit (like SHA-256), note that Grover could find preimages in 2^128 rather than 2^256, which is still fine. Collision-finding doesn’t benefit as much (quantum collision algorithms get 2^(n/3) complexity for n-bit hash, so SHA-256 would have 2^85, which is borderline, but then again one could switch to SHA-384 or SHA-512 in future or rely on the difficulty of forging a full key that matches a commit).
- So, in a scenario where both sides anticipate quantum eavesdroppers, some adjustments can strengthen ASL:
  * Use larger M (size such that even quantum discrete log is hard; unfortunately, quantum DLP is polynomial, so any finite field is breakable at large scale).
  * Or use double encryption: one could combine phi-ns-x with a one-time pad or symmetric cipher as a fail-safe. For instance, after handshake, derive a symmetric key and use a quantum-resistant symmetric cipher (like AES) to encrypt data, and use phi-ns-x mainly for key exchange and authentication. This hybridity might be overkill but it’s a path.
  * More straightforward: rely on the fact that to break ASL fully, the attacker needs to break phi-ns (to get the actual content) or at least the door secrets to forge data. We assume phi-ns stands firm against quantum, so confidentiality remains. For authenticity, ensure some secret piece remains unknown even if states are known. That we have done via door values.

We emphasize that as of writing, quantum computers large enough to

attack 2048-bit discrete log do not exist, but they might within a couple decades. Meanwhile, phi-ns claims resistance due to no known algorithm. It’s newer, so it warrants scrutiny from cryptographers to solidify that claim.

One also should consider **data longevity**: If someone records ASL-encrypted traffic now and stores it, and in 20 years gets a quantum computer, could they decrypt it?
- They would have q (if PK-q) or p (if PK-p) and M, etc. They might break the state progression (trivial with quantum as said) and then they’d know all a_k, but still they’d face the challenge of deciphering each message without p or q. If by then algorithms to invert phi-ns appear (quantum or even classical improvements), then confidentiality would be lost. So, if extremely long-term secrecy (decades) is needed, one might prefer to combine ASL with another PQ algorithm (hybrid encryption) just in case. However, if one believes in the strength of the combinatorial assumption, storing exabytes of classical cipher data might remain safe.

One more angle: lattice or code-based integration: Could phi-ns keys be used in a lattice-like structure to update state instead of discrete exponent? Possibly future research. For now, we keep to classical groups but note the caveat.

In summary, ASL is designed to be post-quantum secure to the best of current knowledge, but like all schemes, that depends on unproven hardness assumptions. We have identified potential quantum weaknesses (like state DLP) and provided mitigations (like secret injection). It is the responsibility of implementers targeting PQ security to enable those features and choose parameters accordingly (e.g., at least Profile L: 256-bit p, q and perhaps deeper recursion or bigger R to frustrate any unknown quantum strategies).

6.5. Unpredictability of States

A crucial security aspect of ASL is that the public values broadcast with each message (the state: u_k and a_k) do not allow an adversary to predict future states or deduce past secrets. This unpredictability ensures **forward secrecy of state** and also contributes to authenticity (an attacker can’t inject a valid future message because they can’t guess the correct state and ciphertext combo).

- The sequence u_k is essentially a pseudo-random sequence from the attackers viewpoint. Without knowing step_u, the best the attacker can do is see it as some random elements in Z_M. Even if they collect many pairs (u_k, u_{k+1}), it doesn’t help them extrapolate unless they solve the discrete log problem to get step_u. As discussed, classically that’s essentially impossible for large M, and even quantumly it’s the one weak

link but mitigated with injection or an alternate approach.
- Because of this, the **probability of an attacker correctly guessing the next valid u_{k+1}, a_{k+1} pair is 1/(M * something)** basically extremely small (like picking a random 2048-bit number correctly).
- Likewise, if an attacker tries to replay an old message, the state (round number or u value) will likely not match the expected next state, and the receiver will detect that. So state sequence provides a built-in anti-replay mechanism. To be safe, implementations should:
  * Track the last seen round or state from each sender.
  * Reject any message that does not have round > last_round (or if round numbers reset, use another mechanism, perhaps window).
  * If round numbers are not explicit and just inferred by order, then any out-of-order arrival should be either ignored or cause a reset. Using explicit numbering is safer to detect replays.

- **Non-deducibility of states** also means even if some state gets revealed (imagine an insider or side-channel reveals the current u_k), the attacker still cannot compute backwards to prior states without step (discrete log again). So backward secrecy is also there: observing a current state doesn't tell you what previous states were used (although an attacker might have seen them anyway if they are public, but if they missed some, they can't derive them from later ones easily).
- If an attacker could somehow find a pattern or relation in the sequence (like a linear relation), that would break unpredictability. But ASL's usage of multiplicative group means it's essentially a power sequence (u_k = u_0 * g^{k*step_u}). This is exponential progression mod M, known to be a sequence with maximal period if g is generator and step_u not a multiple of the order. It's as unpredictible as discrete log assumption. There's no linear relation like LFSRs. So it's solid.
- The inclusion of optional star rotation can make the state not just one sequence but a more complex one: if we modulate or sometimes replace u progression with another factor injection, to an external observer the sequence might look even more random (not just a single exponentiation pattern). However, from a pure unpredictability perspective, a single exponent sequence is already cryptographically random if DLP is hard.

- **No partial prediction**: If an attacker sees u_k and u_{k+2} but misses u_{k+1}, can they compute it? That would require solving something like u_{k+2}/u_k = g^{2*step_u}, thus g^{step_u} would be a square root of that ratio, which is another discrete log essentially (two-step gap doesn't help; it might even give them a slight advantage like they know g^{2*step} now; in some cases they could attempt to take square root if group structure allowed multiple solutions, but since mod prime, taking sqrt is also non-trivial relative to DLP if step is odd etc).
- It's theoretically possible if step_u was small, an attacker could brute force it by checking successive powers. Therefore, step_u should be a large random integer mod (M-1). If one naively set step_u to 1 or a

small number, an attacker could just try those. The specification says step_u is a shared secret and not published, but we should also ensure it’s not too small or guessable. Essentially treat step_u as ~|M| bits of entropy.
- Also if M-1 has small factors and step fell into a subgroup of small order, u might cycle in a shorter loop. That’s an edge case: one should choose M (like a safe prime where M-1 = 2q, q prime, or at least with a large prime factor) and choose step not a multiple of any small factors to ensure full period. Possibly define that step must result in u having large order (maybe even exactly the largest prime factor of M-1).
- If u cycles quickly, predictability might not be easier (still need log, but repetition could allow some statistical detection). Anyway, use large subgroup if not full group.

- During an ongoing session, compromise of secrets provides at most future state knowledge but cannot retroactively decrypt earlier messages due to forward secrecy of ephemeral secrets, if ephemeral were used. Without ephemeral, compromising static key allows decrypting recorded past sessions.

Another aspect: if an attacker can’t predict states, they also can’t perform any kind of chosen-plaintext correlation attack. For example, in some stream ciphers, if state/IV reuse happens, attackers exploit that. Here, state reuse should never happen as long as step isn’t zero mod order or such. There is no key reuse per message because it’s always evolving.

Summing up:
- The state sequence must be treated as a critical part of security. It’s public but pseudo-random. Breaking its unpredictability reduces ASL to a simpler form that might be attackable.
- Implementers must use strong parameters for the state evolution to maintain unpredictability. They must also enforce the protocol rules (no state reuse, detect replays).
- If those are in place, an adversary gains nothing useful from observing states besides synchronization info.

One might wonder if the public nature of a and u could ever leak info about the secret key (p or q). According to the design, they should not: p and q are only present in the encryption transform internally on receiver side. The values u and a are independent of p (they depend on step and g, which are unrelated to p; step is derived from K_s which might indirectly be influenced by p only if something like we derive step from private structure in a fancy implementation, but typically step is handshake ephemeral or from K_s).
If an implementer foolishly derived step_u directly from p or R (like making step = some function of secret key), then u sequence could leak something about the key if someone solved DLP. That is not recommended.

step should be truly random from handshake secret, not deterministic from key. Keep the key and state mechanism separate except the key is needed to decrypt.

This ends the security analysis. Overall, ASL appears to achieve its goals under standard and new hardness assumptions. Nevertheless, it should undergo extensive analysis by the community. As any new cryptosystems, it requires scrutiny, and while we have outlined reasons for its strength, only time and effort will validate them. Users of ASL in critical systems should follow developments and be ready to adapt parameters or schemes if any weaknesses are found.

7. IANA Considerations

This document does not require any actions from IANA. No new protocol numbers, ports, or registries are defined here.

(If ASL were to be assigned a standard port or content type, that could be addressed in a future document. As of now, it operates at an application or session layer with no centralized coordination needed.)

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

[PhiNs] Nsiangani, P. J., "Phi-Ns Cryptosystem: Structured Quadratic Gap Hardness for Public-Key Cryptography", draft-nsiangani-phi-ns-01 (work in progress), February 2026.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

8.2. Informative References

[NISTPQC] National Institute of Standards and Technology, "Post-Quantum Cryptography Standardization Project", 2016-2025.

[ANSSI] Agence Nationale de la Securite des Systemes d’Information, "RGS v2.0 Rfrentiel Gnral de Scurit", 2020.

[NCSC] National Cyber Security Centre, "Cryptographic Recommendations for TLS, VPNs, and Messaging", 2022.

[Lamport] Lamport, L., "Constructing Digital Signatures from a One Way Function", October 1979.

[SHA256] Dobbertin, H., Bosselaers, A., and Preneel, B., "The Hash Functions MD5, SHA-1 and RIPEMD", 1996.

[PhiNs] Nsangani, J., "Phi-Ns: Quadratic Structured Decomposition for Asymmetric Cryptography", Patent Pending, 2025.

Author’s Address

Parfait Junior Nsiangani (Editor)
Email: jnsiangani@gmail.com

Figure 1: Original draft content (imported)

2. References

Author’s Address

P.J. Nsiangani (editor)
ACE Working Group