Nitrous Oxide

A *Wicked-Fast* Post-Quantum Tunnel

v0.8-11-ge7c3df60f414

Jason A. Cooper[1] [0009-0005-5525-9647]

Independent Researcher
jason@coldbeach.io
https://coldbeach.io

Abstract. Nitrous Oxide is a modern, fast, rock-solid cryptographic state machine. It is safe from quantum computing store-now-decrypt-later attacks. There are no static keys, nor static peer attributes. The N2O protocol maintains cryptographic state synchrony for years, regardless of packet loss or interference. Future sessions are negotiated *within* the current session. The post-quantum KEM (Key Encapsulation Mechanism) is *only* needed for the first session, called the INTRODUCTION. When the INTRODUCTION is performed in close physical proximity, no public keys are *ever* exposed on the untrusted Internet. Data to be encrypted by N2O *never* waits for any session negotiation. If the network is up, data is immediately encrypted and sent.

Keywords: post-quantum · KEM · AEAD · key continuity · ephemeral · asynchronous · network · protocol · FIPS

1 Introduction

Networks are a battlefield. There's no such thing as routine. Congestion and RF interference cause packet drops. Routing changes and NAT timeouts leave half-open sockets. CG-NAT makes it damn near impossible to reach a peer directly.

It's hard enough to maintain a connection over a network for any length of time. Harder yet to keep a shared cryptographic state in sync across the same environment.

Yet, for humans, we don't start with a blank slate every time we get together. We are introduced, become familiar with one another, and recognize each other later. Over decades, we build up memories each time we gather; our shared experiences bind us.

Ideally, cryptography enables communication with our peers over vast distances, through untrusted and hostile networks. A well designed cryptographic system should give us confidence that we are speaking to the peer we know. No one else is listening in, nor modifying our conversation.

Nitrous Oxide is a concise cryptographic protocol; it ensures you're communicating with peers you know, regardless of dropped packets, latency, or changing routes. N2O builds on top of the lessons learned from decades of widely deployed encryption protocols.

2 Why N2O?

N2O provides the highest level of security, out-of-the-box. It runs baremetal, on small microcontrollers like a Cortex-M4, protecting bare RF links and serial connections on printed circuit boards. N2O is power-efficient, it only does work when the device needs to send or receive data. Remote, isolated installs can be reached with confidence. Even at rates of one message per *month*.

N2O runs on phones, servers, and laptops. It integrates with existing network stacks and VPN APIs. N2O doesn't timeout, and doesn't lose state sync. When a message is decrypted by a peer context, you can be certain the message is genuinely private, and from that session contexts' peer.

While covering all these usecases, N2O provides post-quantum security, ephemeral key exchange, forward secrecy, and man-in-the-middle (MitM) mitigation.

3 Overview

Nitrous Oxide works differently than other cryptographic protocols. TLS [7][2], OpenVPN[3], and Wireguard[9] setup a new context with each network connection. N2O, on the other hand, performs an INTRODUCTION once, and maintains the resulting context over many connections. Conceptually, N2O is similar to "adopting a device" in smart home ecosystems. In this way, N2O bears resemblance to ephemeral protocols such as ZRTP[28]. With appropriate session rotation, an N2O peer context will last for years.

When reading this document, we say "context" for two different objects. N2O has a *peer* context, which holds two *session* contexts. See the recommended layout in Figure 1.

3.1 Ephemeral Protocol

N2O is an ephemeral protocol. It *only* uses ephemeral keys, there are no static keys, nor certificates. The identity of a peer is established in one of two ways. The most common is by performing the INTRODUCTION in physical proximity. The second is by relying on a mutually trusted peer to relay an INTRODUCTION. Attributes of identity, such as email address, hostname, etc are asserted dynamically, as requested. This assertion is performed by a higher level protocol, to be defined separately. For the purpose of this document, each peer assigns identity attributes to the N2O peer context after INTRODUCTION.

All N2O private keys, public keys (PEER KEYS), and seeds are immediately disposed of after the session context is successfully negotiated. No long term (spanning more than one session) secrets are ever retained. This further frustrates store-now-decrypt-later because there are never any secrets laying around which could decrypt old sessions.

```
pub struct N2O {
   peers: HashMap<SessionID, PeerContext>,
}
pub struct PeerContext {
   provenance: Provenance,
   curr: SessionContext,
   next: SessionContext,
   identifiers: HashMap<typ, id>,
}
pub struct SessionContext {
   data_init: Key,
   data_resp: Key,
   nonce_init: Nonce,
   nonce_resp: Nonce,
   sess_id_init: SessionID,
   sess_id_resp: SessionID,
   seed_inf: Seed,
   auth_str: AuthStr,
}
```

Fig. 1. Rust struct declaration of Peer contexts and Session contexts

3.2 Constrained Devices

During the INTRODUCTION, N2O uses the post-quantum KEMs, ML-KEM[21], or CRYSTALS-Kyber[16]. On a common microcontroller, such as the Cortex-M4, these KEMs run, but very slowly. The design of N2O gracefully handles this by *only* requiring the KEM during the INTRODUCTION phase.

N2O is designed to maintain cryptographic context for years. Therefore, we can use FIPS ML-KEM[21] at its highest strength, ML-KEM-1024 [22] and non-FIPS Kyber[16] at KYBER-1024[17], with no impact on day-to-day performance.

The *minimum* capable CPU is an ARM Cortex-M4. Any processor of equal or greater capability is suitable. 32 bit, 64 bit, big endian or little endian. ARM, MIPS, x86, or other mainstream ISAs supported by the predominant open source compiler projects.

3.3 Shared Entropy

Microcontrollers are notorious for their lack of entropy, especially on first boot. N2O is resilient to this frequent condition due to the use of a bidirectional-KEM during INTRODUCTION. The bidirectional-KEM passes a 256 bit seed from each peer to the other. Both seeds are used to generate the first shared secret. Thus, if one of the peers is entropy deficient, N2O can still generate a strong shared secret. Additionally, the ephemeral KEM keypair created by the microcontroller on first boot is assumed weak as well. Thus, the bidirectional-KEM is necessary to ensure that the shared secret is unguessable.

This entropy-sharing concept persists throughout the lifetime of an N2O context. Each session context negotiation uses a fresh 256 bit seed from each peer, as well as a seed from the previous session, called `seed`$_{\infty}$. The `seed`$_{\infty}$ is the concept of Key Continuity borrowed from ZRTP[29]. After each session negotiation, a separate output of the key derivation function is used to stir the entropy pool of the RNG. See the `chum` in Section 8.2.

3.4 Context Provenance

N2O uses PROVENANCE binding to inform the user of the origins of the context. The PROVENANCE records who, what, where, when, and how about the INTRODUCTION. For N2O, the "what" serves two purposes. The first is to identify the specific cryptographic protocol with the algorithm string, `algo`. For example:

```
N2O-V1-MLKEM1024-SHA512-AES256-GCM
```

Note that "`V1`" is the version of the protocol, not of this paper, nor of any source code implementation. A complete PROVENANCE looks like:

```
N2O-V1-KYBER1024-BLAKE3-CHACHA20-POLY1305|\
DIRECT - Bluetooth|\
I: Alice Cooper (phone)|\
R: Bob Ross (phone)|\
MXQ4+P5 New York City, USA; Earth|\
1721918909
```

Whenever feasible, this information is presented to both users, and visually confirmed by both prior to completing INTRODUCTION. It is *not* identity. The best way to understand it is through a scenario in the future. Bob asks himself, "where did I meet Alice Cooper?", and the N2O app can pull up the information from the PROVENANCE to refresh his memory. We're just answering the question, "Where did this context come from?" The PROVENANCE is also used as an immutable (once agreed upon) domain separator for all session context derivations.

This also gives N2O the means to detect when a nearby peer is using a weaker INTRODUCTION than is now possible, eg:

```
N2O-V1-KYBER1024-BLAKE3-CHACHA20-POLY1305|\
RELAY - N2O|\
I: Alice Cooper (phone)|\
IV: DIRECT - Bluetooth|\
V: Fred Johnson (phone)|\
VR: DIRECT - NFC|\
R: Bob Ross (phone)|\
...
```

and can prompt Alice if she'd like to redo the INTRODUCTION with Bob directly.

Constructing the PROVENANCE this way enables N2O to provide the user with a clean, deterministic UI hint as to the confidence level of a given INTRODUCTION.

A critical piece of the provenance is the app putting together the PROVENANCE for the user, showing the user, and the user confirming the information. Additionally, the responder app must also show the PROVENANCE to the other user for confirmation prior to continuing with the INTRODUCTION. See Figure 2 to get an idea of the user experience for this type of INTRODUCTION.

In non-user communications, like VPNs, or microcontroller-based devices, the PROVENANCE is simply a domain separator. As long as it contains `algo` and information unique to the INTRODUCTION, all is well. See Section 8.2.

3.5 Resilient Session Negotiation

In difficult RF environments, bidirectional communication can be spotty and brief. To overcome this hurdle, N2O immediately begins negotiating the *next* session context as soon as it detects bidirectional data transfer within the *current* session. This negotiation *always occurs within* the current session, drastically reducing information available to an outside observer.

N2O takes the traditional concept of session negotiation and splits it into two discrete parts: SESSION NEGOTIATION and SESSION MIGRATION. By doing so, N2O can perform SESSION NEGOTIATION *within* the current session, *before* SESSION MIGRATION is needed.

As SESSION NEGOTIATION now occurs entirely within the symmetric encryption of the current session, it can be reduced to the simplest cryptographic operation: the exchange of two seeds.

3.6 No Timeouts

Any limits expressed in N2O are in terms of bytes, packets, or counters. There are deliberately no timeouts. If there is no network connectivity, or only communication in one direction, there's no reason to break cryptographic sync. The N2O state machine is infinitely patient.

3.7 Scope

Any two electronic devices, Cortex-M4 or more capable, can be peers. Any electronic medium, serial or framed, direct or packet switched is suitable. Any data which can be represented in binary is an acceptable payload.

Any distance is acceptable, including inter-planetary. As mentioned in subsection 3.6, the N2O state machine is patient, and will continue to send and receive data, regardless of latency.

3.8 Out of Scope

Abstract devices, which a user has *no* possibility of being in physical proximity to, are **not** acceptable peers. Examples of unsuitable peers are the cloud servers hosting `google.com`. A user has no expectation of ever being able to lay a hand on a server, virtual machine, or container, owned by google.com and verify "This is the instance I'm connecting to when I connect to Google."

However, an IT administrator at Google *does* have the possibility of laying hands on individual servers. Therefore, N2O would be appropriate for the administrator to use for connecting to each individual server.

This is a high level way of saying "Don't pass N2O contexts from one device to another." In your deployment of N2O, if you have to pass a context from one device, virtual machine, or card to another, you need to reach out for guidance. N2O contexts are concrete, not abstract. Abstract identities are handled at a layer above N2O by groups of devices. A user might have four devices, or a company might have hundreds of users. These are groups. N2O supports those use cases, by deliberately letting other layers handle it.

In short, N2O is *not* intended to replace TLS.

3.9 Threat Model

All passive outside attackers are in-scope. Both store-now-decrypt-later, as well as observe and decrypt. Additionally, tracking peers across time and networks is in-scope.

All active outside attackers are in-scope. Replay attacks, modify and replay attacks, drop packets, length extensions, and truncations.

All passive inside attackers are in-scope. Specifically, attackers who can observe data to be encrypted and sent over the wire.

Most active inside attackers are in-scope. For example, attackers who can control data to be encrypted by N2O are in-scope.

The only attackers who are out-of-scope, are attackers who have access to the N2O context. Once the attacker knows the private context information, it's game over.

4 Contributions

4.1 PQ Ephemeral Key Agreement

Ephemeral key agreement protocols, such as ZRTP[28], have traditionally relied on some form of Diffie-Hellman[8] key agreement with ephemeral key pairs (DHE). Used alone, DHE systems are directly vulnerable to *active* man-in-the-middle attacks. ZRTP[28] uses three features to detect, and thus thwart, MitM attacks: Diffie-Hellman[8], Key Continuity[29], and a Short Authentication String[30].

Unfortunately, the only key agreement primitive selected by the NIST PQC[24] is a Key Encapsulation Mechanism (KEM), ML-KEM[21]. When one uses a bare ephemeral KEM for key agreement, it suffers even worse than DHE[8] systems during MitM, because the attacker learns the complete shared secret encapsulated in the KEM. The attacker can then *passively* decrypt sessions undetected.

To force an active attacker who attempts to MitM the INTRODUCTION to also *actively* MitM the resulting session, N2O requires that both peers include the PEER KEYS in the first session context derivation. This results in different shared secrets if there was a MitM during INTRODUCTION. To avoid detection, the attacker *must* attack the INTRODUCTION **and** the resulting session, which is typically on a completely different network path.

Now that N2O forces a MitM to continue being present in the session, Key Continuity[29] and the Short Authentication String[30] can be used to achieve equivalent confidence of ZRTP[28] that no MitM is present after INTRODUCTION. See the Section 6 "MiTM Detection" for more details.

4.2 Peer Keys

KEMs are young. The NIST standard, FIPS 203[21], was published on the 13th of August, 2024. It will take several years of widely deployed use, and cryptanalysis, before we can use it like other established primitives. To mitigate the risk of a young primitive, N2O uses KEMs, but doesn't fully trust them. N2O uses the PROVENANCE to remember how the PEER KEYS (public keys) were exposed, if at all.

If a peer is seen over a more secure path than the original INTRODUCTION, N2O can trigger a reintroduction over the more secure path. This enables users to make an N2O INTRODUCTION when they need to, and make it stronger when occasion permits.

It's helpful to imagine PEER KEYS like children. Parents are cautious about where their children go, who they spend time with, and how much of the Internet they're exposed to. PEER KEYS are the same. We hand them to people we trust, but we don't let them run loose in the city at 2am.

We refer to "public keys" as PEER KEYS to draw attention to the difference in how N2O treats them. With this design, we can use KEMs, while hedging against the most likely failure modes.

4.3 Asynchronous Session Negotiation

Many poor user experiences with VPNs occur when a user device wakes up and attempts to communicate through a VPN. The VPN client wakes up, and attempts to negotiate a new session over half-open sockets, stale routes, or one-way RF comms.

N2O mitigates most of this problem by making session negotiation asynchronous from session migration. With the session keys already in hand, an N2O peer can immediately encrypt and send the data packet. After INTRODUCTION, applications shall never wait for a session negotiation or migration.

4.4 Concealed Session Negotiation

Since session negotiation is asynchronous, N2O can perform the negotiation for the *next* session *within* the *current* session. Never exposing session negotiation to the hostile Internet greatly improves N2O security posture, reduces risk, and simplifies the design. N2O session negotiation is simply the exchange of two 32 byte seeds.

4.5 PROVENANCE Binding

Nitrous Oxide peer contexts, like human relationships, are long lived. To record the environment surrounding the INTRODUCTION, we encode information about the event into the PROVENANCE. The PROVENANCE is then asserted in every session context generation performed for the lifetime of the N2O peer context.

The PROVENANCE records who, what, where, when, and how about the INTRODUCTION. This is *not* identity. The purpose of PROVENANCE binding is to inform the owners of the context as to the provenance of the peer context. Conveniently, this provides clean domain separation, not just per-protocol, but also per-N2O-peer-context. This applies in all of the KDFs used to generate secrets.

Properly constructed, the PROVENANCE provides a deterministic UI hint as to the level of confidence in the INTRODUCTION. INTRODUCTIONs performed in physical proximity have higher confidence than those performed via relay or remotely.

5 INTRODUCTION Workflows

Performing an INTRODUCTION occurs one of two ways. Either the user is physically close to the new peer, or the user leverages an existing peer relationship to perform the INTRODUCTION. See Figure 5 for an overview of the messages and actions performed by both peers.

N2O peers do **not** listen for INTRODUCTIONs by default. Similar to modern Bluetooth pairing, N2O peers only listen for INTRODUCTIONs after deliberate user action. And only for a short period of time after.

5.1 Physical Proximity

When a user is in physical proximity to the new peer, the INTRODUCTION can be performed over any direct, short range medium. This includes, but is not limited to:

- Bluetooth
- NFC
- Phone camera / display
- Zero TTL local IP network
- Copper wire (USB, serial, I2C, SPI, UART, etc)

An example sequence diagram, including user flows, can be found in Figure 2. This follows very closely to modern Bluetooth discovery, where discovery is *only* enabled when the user opens the appropriate settings screen. Note that unsecured messages are **red**, and messages secured by N2O are **blue**. **Green** messages are secured by non-N2O encryption. Gray messages indicate interactions between human users and their respective devices.

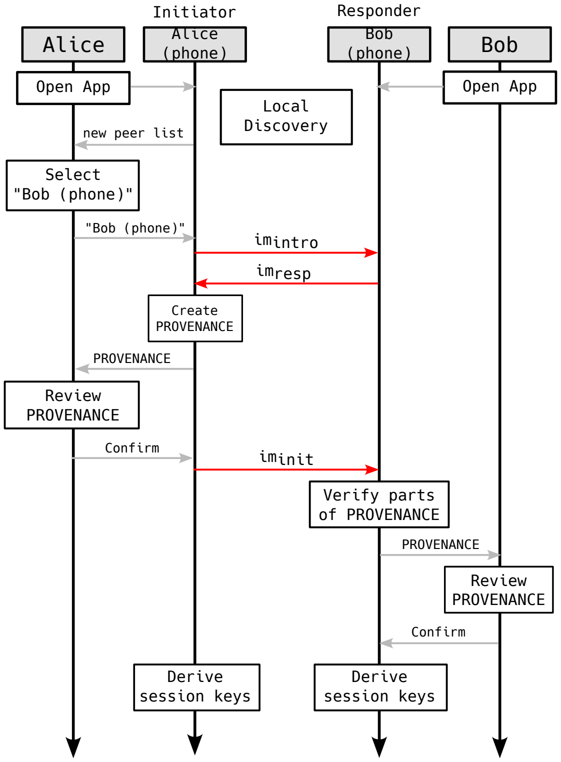

Fig. 2. DIRECT (physical proximity) INTRODUCTION sequence diagram

5.2 Mutual Established Peer

When a user wishes to establish an N2O peer context with a remote peer, it requires a mutual peer with a secure connection to both peers.

With our traditional characters, Alice wants to communicate with Charlie via N2O, but Charlie is far away. Previously, Alice established a context with Bob, whom she works with. Bob has an established N2O context with Charlie, who he met last year at a conference.

In this scenario, Bob will be our mutual established peer. Alice will initiate the INTRODUCTION to Charlie via Bob. All messages for the INTRODUCTION will be contained within the existing Alice-Bob N2O context, or the Bob-Charlie N2O context.

From a human perspective, we are modeling Bob introducing Alice to Charlie at Alice's request. See Figure 3 for a high level overview of the conversation flow. Unlike the physical proximity scenario, all messages are **blue** because they are secured by pre-existing N2O contexts.

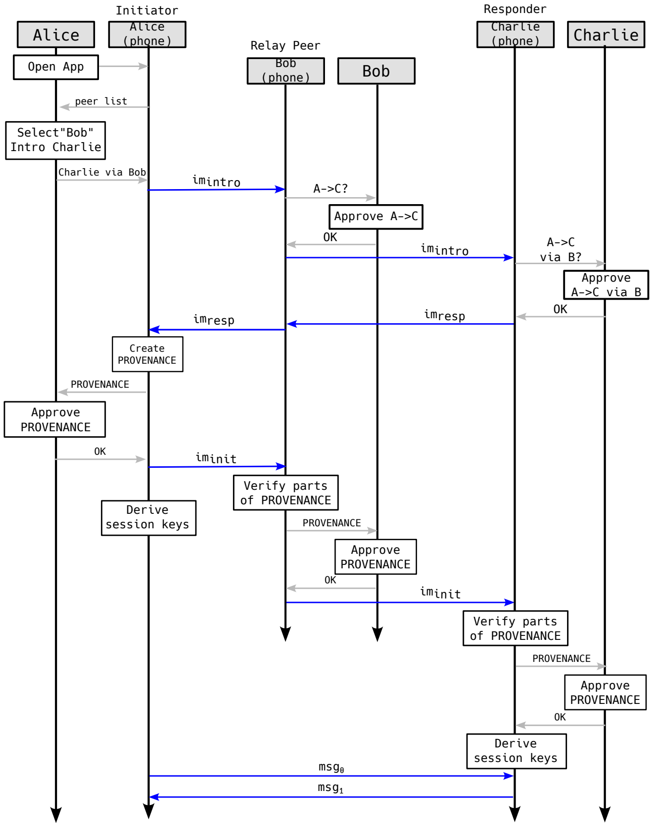

Fig. 3. RELAY (trusted peer) INTRODUCTION sequence diagram

5.3 Existing Non-N2O Peer

On some occasions, existing N2O contexts may not exist. In this case, other existing secure channels may be used to relay the INTRODUCTION. Some examples include SSH[26], TLS[7][2], or other VPNs. In mesh VPNs such as Tailscale[25] or Netbird[5], the INTRODUCTION can be handled in a similar manner to how those tools currently pass Wireguard[9] public keys. See Figure 4.

6 MitM Detection

N2O is an ephemeral key system, conceptually, very similar to ZRTP[28]. Raw ephemeral key protocols are susceptible to active MITM attacks where the attacker, Eve, swaps out public keys in a way which makes Eve the peer to both Alice and Charlie. Eve can then view and relay all plaintext messages between Alice and Charlie.

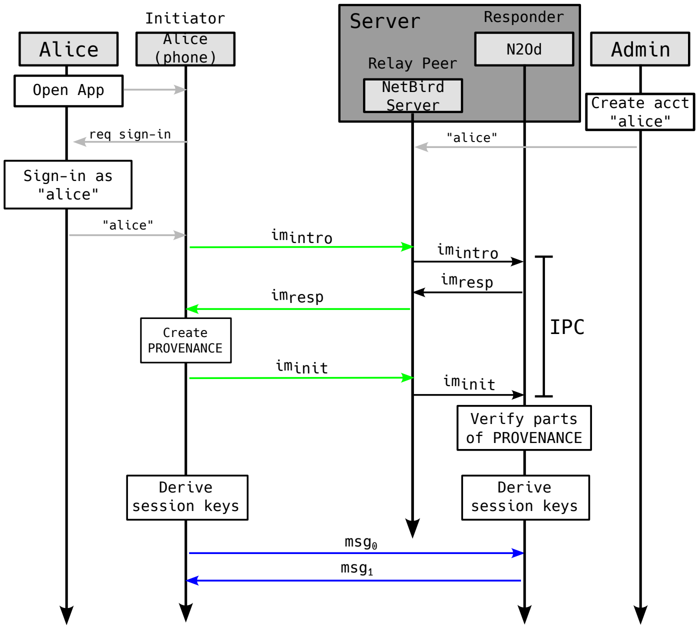

Fig. 4. RELAY (non-N2O peer) INTRODUCTION sequence diagram

6.1 Prerequisites

Prerequisites for a Relay INTRODUCTION:

1. Alice would like to talk with Charlie, but they aren't in physical proximity
2. Alice has an existing N2O peer context with Bob, they work together
3. Bob has an existing N2O peer context with Charlie, they met at a conference last year
4. Alice will ask Bob to introduce Charlie and her via N2O
5. After this Relay INTRODUCTION, Alice and Charlie will be able to communicate directly via N2O
6. While Bob, the human, is trusted, we presume Eve is inside Bob's device
7. Dale is late to the party, and attempts to read a future N2O session negotiation without being present during INTRODUCTION

6.2 ZRTP MitM Detection

Let's first look at how ZRTP[28] handles this. ZRTP[28] mitigates this MITM attack with three tools:

- Diffie-Hellman[8] (DH in DHE)
- Key Continuity[29]
- Short Authentication String[30]

DHE uses Alice and Charlies public keys when deriving the shared secret. Eve *must* substitute the DHE public keys, and thus, any further MITM attack she performs *must* be an active MITM because the public keys are used in the calculation of the shared secret.

Key Continuity[29] uses some secret material, generated from the previous session secret, in the next sessions key schedule. When Dale comes along later, after the initial handshake, and attempts a MitM, it will fail. Dale has no way to learn the secret material from the previous session. Therefore, Dale *must* be present during the first handshake, or he'll never be able to successfully MitM the session.

Additionally, if Eve successfully MitM'd the first handshake, she *must* continue to MitM *every* session from then on, else she'll be detected due to decryption failure.

The Short Authentication String[30] (SAS) is mnemonic representation of a small amount of data derived from the current session shared secret. If Alice and Charlie have the same shared secret (no MitM), then the SAS will also be the same. Verifying the SAS out-of-band gives confidence that there is no MitM in their session.

It should be noted that ZRTP[28] has been widely deployed for many decades, and although not quantum-safe, it has proven robust against MITM attacks. Second, ZRTP[28] key agreements occur over the open Internet, thus are much more exposed to attacks than n2o.

6.3 N2O MitM Detection

KEMs are the only post quantum algorithm accepted by NIST for key agreement. Unfortunately, they're a Key Encapsulation Mechanism (KEM). There is no defined way to use it similar to Diffie-Hellman[8].

The most important aspect of Diffie-Hellman[8] is that an active MitM present during the initial handshake alters the shared secret differently for each peer. Thus, the attacker *must* remain actively attacking the first session as well, otherwise the decryption will fail, and the attacker will be detected.

In order for Key Continuity[29] and the Authentication String[30] to effectively detect a MitM, the n2o introduction **must** have this same property as Diffie-Hellman[8].

n2o achieves this by including *both* peer keys in the first session KDF. Thus, when the first shared secrets are calculated, they will be different between the peers if a MitM was present. This forces Eve to continue her active MitM attack against the first session as well. Otherwise, decryption will fail, and Eve will be detected.

Even though n2o doesn't have public key exchanges for sessions, an active inside attacker, similar to Dale, could view the network traffic from a session negotiation via a coding error and a malicious packet sent through the tunnel. This could give Dale knowledge of the session negotiation $\texttt{seed}_{\texttt{init}}$ and / or $\texttt{seed}_{\texttt{resp}}$ as they are sent over the encrypted session. n2o uses Key Continuity[29] to render this knowlege useless to the attacker. This secret material from the previous session is *never* sent over the wire, and therefore, not visible to Dale.

Additionally, Key Continuity[29] forces Eve, the MitM present during the INTRODUCTION and the first session, to continue to be present for *all* future sessions. Else decryption will fail, and she'll be detected.

Should Eve manage to maintain her MitM attack against every session over every network path, attempting to verify the Authentication String[30] will reveal her presence.

7 Cryptographic Protocol

There are two modes of operation. The peer is either running FIPS-certified mode, or non-FIPS mode. There is deliberately *no* interoperability between the two modes. In fact, from the user perspective, they are two different apps, or ideally, two different devices.

For FIPS-mode, the following algorithm string identifier, `algo` is used in the PROVENANCE:

```
N2O-V1-MLKEM1024-SHA512-AES256-GCM
```

The cryptographic primitives for FIPS-mode are selected to meet or exceed CNSA 2.0[1].

For non-FIPS mode, N2O uses:

```
N2O-V1-KYBER1024-BLAKE3-CHACHA20-POLY1305
```

for its algorithm string. This mode still requires a cryptographically secure random number generator, but is more tolerant to momentary weaknesses in the RNG. While there is no magic way to create security from insecure randomness, this mode is more tolerant of real world conditions.

7.1 KEM

In both modes of operation, N2O uses 1024bit KEM. For FIPS mode N2O uses ML-KEM-1024[21]. Non-FIPS mode uses Kyber-1024[16].

FIPS 203, Appendix C[23] contains an overview of the algorithm changes from CRYSTALS-Kyber submission 3[16] to the final FIPS publication. The four changes are:

1. Shared secret key is fixed to 256 bits / 32 bytes
2. ML-KEM.Encaps() doesn't include a hash of the ciphertext in the derivation of the shared secret
3. ML-KEM.Encaps() removes a hash of the initial randomness as NIST places standards on randomness generation, making it unnecessary
4. An explicit check was added to ML-KEM.Encaps() to ensure the encapsulation key decodes to an array of integers mod q without reductions

Please see Appendix C of FIPS 203[23] for details. N2O already assumes a shared secret (`seed`) of 32 bytes.

In non-FIPS scenarios, N2O uses Kyber-1024[16] due to it hashing the initial randomness in ML-KEM.Encaps(). The security of both KEMs rely on uniformly distributed randomness in the encapsulated shared secret (`seed`). In the absence of a FIPS-certified random number generator, Kyber[16] is the safer choice.

7.2 Key Derivation

N2O deliberately uses an open approach to key derivation. It consists of two steps:

$$\begin{aligned}
\texttt{hkey} &= \text{HASH}(\texttt{secret} \,||\, \texttt{asserts}) && (1)\\
\texttt{okm}_1 &= \text{HMAC}(\texttt{hkey},\ \texttt{INFO}_1,\ \texttt{len}_1) && (2)\\
\texttt{okm}_2 &= \text{HMAC}(\texttt{hkey},\ \texttt{INFO}_2,\ \texttt{len}_2) && (3)\\
&\ldots\\
\texttt{okm}_n &= \text{HMAC}(\texttt{hkey},\ \texttt{INFO}_n,\ \texttt{len}_n) && (4)
\end{aligned}$$

Across N2O, all strings shall *not* be `NULL` terminated. If you are implementing in a language with `NULL`-terminated strings, **you** are responsible for not including the `NULL` terminator.

`secret` is the shared secret. `asserts` contains all the shared values which we want to assert haven't been altered, but aren't secret from a cryptographic perspective. `hkey` is a 512 bit, 64 byte uniformly distributed shared secret. $\texttt{okm}_n$ is the key we're creating. $\texttt{INFO}_n$ contains a domain-separating string for a desired key, and $\texttt{len}_n$ is the length of the output $\texttt{okm}_n$ in bytes.

FIPS-mode For FIPS-mode, N2O uses `SHA-512`[19] for HASH() and `HMAC-SHA-512`[20] for HMAC() as a pseudorandom function for its KDF. If the desired $\texttt{len}_n$ of $\texttt{okm}_n$ is less than 512 bits, then trunc() is used:

$$\texttt{okm}_n = \text{trunc}(\texttt{okm}_n,\ \texttt{len}_n) \qquad (5)$$

non-FIPS-mode For non-FIPS-mode, we use `BLAKE3`'s[15] `hash`() for HASH(), and `derive_key`() for HMAC(). Blake3[15] is fast, even on 32 bit systems, and highly parallelizable. Additionally, it provides a variable output length, if needed.

7.3 AEAD

N2O expects the standard AEAD signatures:

$$\begin{aligned}
\texttt{ciphertext} &= \text{AEAD\_enc}(\texttt{dkey}, \texttt{nonce}, \texttt{add\_data}, \texttt{data}) && (6)\\
\texttt{data} &= \text{AEAD\_dec}(\texttt{dkey}, \texttt{nonce}, \texttt{add\_data}, \texttt{ciphertext}) && (7)
\end{aligned}$$

where `ciphertext` is the result of encrypting the `data` and appending the authentication tag. `dkey` is the symmetric key derived from the KDF, and `nonce` is the derived nonce that is incremented after each call to AEAD. `add_data` is the additional data transmitted in the clear, and included under the authentication tag. For N2O, `add_data` shall include the entire N2O plaintext header. Last, `data` is the message to be encrypted and sent to the peer.

Note that N2O is *directional*. When the Initiator is encrypting a message to send to the Responder, the Initiator uses $curr.\mathtt{dkey}_{\mathtt{init}}$ as the AEAD `dkey`, and $curr.\mathtt{nonce}_{\mathtt{init}}$ as the `nonce`. The Responder then reconstructs the $\mathtt{nonce}_{\mathtt{init}}$ for the message from the message `counter` and its $curr.\mathtt{nonce}_{\mathtt{init}}$. The Responder uses the same $\mathtt{dkey}_{\mathtt{init}}$ from its *curr* session context to decrypt the message.

N2O does *not* use a "key committing AEAD" such as Chacha20-Blake3[12] or AES-GEM[4]. There are some proposals for such a primitive, but none have been standardized, nor seen mainstream adoption. As such, N2O protects against partitioning oracle attacks[13] in two ways. First, N2O uses the nonce as a counter for both the cryptographic unique-value-per-message requirement, **and** the protocol counter requirement. If the same counter value is seen by the protocol stack, it *must* be silently dropped. Additionally, if an AEAD decryption fails, it *must* be silently dropped. By **always** failing silently, the attacker gains a lot less information than ordinarily possible.

Indirectly, N2O also protects from partitioning oracle attacks[13] by *never* deriving key material from passwords, and always using randomness (`seeds`) from **both** peers to construct each session secret.

Both FIPS and non-FIPS modes use a 256 bit key and a 96 bit nonce. Both modes permit the nonce to be a per-message counter.

FIPS-mode For FIPS-mode, N2O uses `AES-256-GCM`[11] as its AEAD. Using a standard construction of AES[18] meets FIPS requirements, and using a 256bit symmetric key meets CNSA 2.0[1] requirements.

non-FIPS-mode For non-FIPS mode, N2O uses `chacha20-poly1305`[14] as published by the IRTF. This is a very widely deployed and reviewed primitive. It is the AEAD primitive used in Wireguard[9], as well as TLS[7][2].

8 Protocol Design

8.1 Properly Seeded CS-PRNG

The following pseudocode shall place `size` bytes into a buffer called `seed` from a properly seeded CS-PRNG:

$$\mathtt{seed} = \text{getrandom}(\mathtt{size}) \tag{8}$$

$$\mathtt{void}\ \text{add\_entropy}(\mathtt{buf}, \mathtt{blen}) \tag{9}$$

And the opposite operation, add_entropy() is a non-entropy-crediting function which adds `buf`, of length `blen` to the entropy pool.

8.2 INTRODUCTION Session Negotiation

Bootstrapping an N2O context is very similar to future session negotiations, with a few critical differences.

1. KEM keys are used to protect the seeds
2. The PROVENANCE is generated and verified
3. Key derivation uses the PEER KEYS vice the $\texttt{seed}_\infty$
4. The PROVENANCE is sent in message $m_{\texttt{init}}$

See Figure 5 for an overview of how INTRODUCTION works. Note that many details are left out of the diagram to enhance readability. Please refer to the following sections for detailed descriptions.

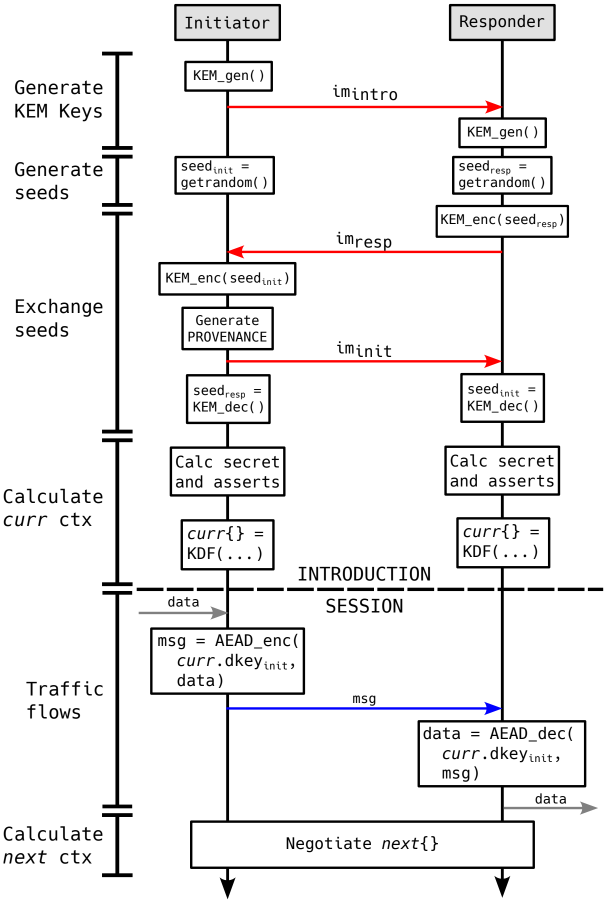

Fig. 5. INTRODUCTION sequence diagram

At any given time, the peer context will only *ever* have two session contexts, *curr*, and *next*. The INTRODUCTION calculates and fills the *curr* session context. All future session negotiations calculate and fill the *next* session context. During session migration, N2O swaps *next* into *curr* context and ZEROES out the previous *curr* context. See Section 8.6 for details on session migration.

Generate KEM Ephemeral Keys The Initiator generates the ephemeral KEM keys for the INTRODUCTION:

$$(\texttt{priv}_{\texttt{init}}, \texttt{peer}_{\texttt{init}}) = \text{KEM\_gen}(\&\text{getrandom}()) \tag{10}$$

The Responder generates its keypair:

$$(\texttt{priv}_{\texttt{resp}}, \texttt{peer}_{\texttt{resp}}) = \text{KEM\_gen}(\&\text{getrandom}()) \tag{11}$$

Message $\texttt{im}_{\texttt{intro}}$ The Initiator uses its PEER KEY to construct and send message $\texttt{im}_{\texttt{intro}}$:

$$\texttt{im}_{\texttt{intro}} = \texttt{peer}_{\texttt{init}} \tag{12}$$

Generate Seeds Both peers generate their respective 32 byte, 256 bit seeds. First, the Initiator:

$$\texttt{seed}_{\texttt{init}} = \text{getrandom}(32) \tag{13}$$

Then, the Responder does the same:

$$\texttt{seed}_{\texttt{resp}} = \text{getrandom}(32) \tag{14}$$

Message $\texttt{im}_{\texttt{resp}}$ After the Responder receives message $\texttt{im}_{\texttt{intro}}$ it encrypts its $\texttt{seed}_{\texttt{resp}}$ using the Initiator's PEER_KEY, $\texttt{peer}_{\texttt{init}}$.

$$\texttt{enc\_seed}_{\texttt{resp}} = \text{KEM\_enc}(\texttt{seed}_{\texttt{resp}}, \texttt{peer}_{\texttt{init}}) \tag{15}$$

The Responder creates and sends message $\texttt{im}_{\texttt{resp}}$:

$$\texttt{im}_{\texttt{resp}} = \texttt{enc\_seed}_{\texttt{resp}} \,||\, \texttt{peer}_{\texttt{resp}} \tag{16}$$

Prepare PROVENANCE Upon receipt of message $\texttt{im}_{\texttt{resp}}$, the Initiator puts together the PROVENANCE:

```
N2O-V1-KYBER1024-BLAKE3-CHACHA20-POLY1305|\
DIRECT - Bluetooth|\
I: Alice Cooper (phone)|\
R: Bob Ross (phone)|\
MXQ4+P5 New York City, USA; Earth|\
1721918909
```

Line 1 is the algorithm string, `algo`. Line 2 is the manner and medium of the INTRODUCTION. Lines 3 and 4 are the peers involved in the INTRODUCTION. Line 5 is the Open Location Code followed by the planet where the INTRODUCTION occurred. Last, Line 6 is the number of seconds since midnight, 1 January, 1970, UTC.

Before sending the PROVENANCE, the Initiator shows a human-readable summary of the PROVENANCE to the user. Once the user verifies the PROVENANCE, the Initiator can continue.

As mentioned previously, the PROVENANCE is useful to show both humans, *if* two humans are involved. Otherwise, the PROVENANCE may be auto-generated and used as further domain separation and a reference to detemine if a more secure path is available to perform INTRODUCTION more securely.

The Initiator saves the PROVENANCE for the life of the context with the peer.

Message $\mathtt{im}_{\mathtt{init}}$ The Initiator encrypts its $\mathtt{seed}_{\mathtt{init}}$ using the Responder's PEER KEY, $\mathtt{peer}_{\mathtt{resp}}$:

$$\mathtt{enc\_seed}_{\mathtt{init}} = \mathrm{KEM\_enc}(\mathtt{seed}_{\mathtt{init}}, \mathtt{peer}_{\mathtt{resp}}) \tag{17}$$

The Initiator creates and sends message $\mathtt{im}_{\mathtt{init}}$:

$$\mathtt{im}_{\mathtt{init}} \;=\; \mathtt{enc\_seed}_{\mathtt{init}} \,||\, \mathrm{PROVENANCE} \tag{18}$$

The Reponder verifies the relevant pieces of the PROVENANCE, and asks the user to confirm the PROVENANCE. The Responder then saves the PROVENANCE for the life of the context with the peer.

Decrypt Seeds Both peers then decrypt their received seeds. First, the Initiator does so:

$$\mathtt{seed}_{\mathtt{resp}} = \mathrm{KEM\_dec}(\mathtt{enc\_seed}_{\mathtt{resp}}, \mathtt{priv}_{\mathtt{init}}) \tag{19}$$

Followed by the Responder performing the equivalent:

$$\mathtt{seed}_{\mathtt{init}} = \mathrm{KEM\_dec}(\mathtt{enc\_seed}_{\mathtt{init}}, \mathtt{priv}_{\mathtt{resp}}) \tag{20}$$

Once the seed is decrypted, each peer shall ZERO out the KEM `priv` keys.

Prepare `secret` and `asserts` In order for the Initiator and the Responder to both arrive at the same shared session context, they each must concatenate the exact same values in the exact same order. Described here:

$$\mathtt{secret} = \mathtt{seed}_{\mathtt{init}} \,||\, \mathtt{seed}_{\mathtt{resp}} \tag{21}$$

$$\mathtt{asserts} = \mathtt{peer}_{\mathtt{init}} \,||\, \mathtt{peer}_{\mathtt{resp}} \,||\, \mathrm{PROVENANCE} \tag{22}$$

This is the critical piece that enables ephemeral KEM to achieve the same security posture as non-PQ ZRTP[28]. The PEER KEYS *must* be included in the `asserts`.

Calculate First Session Keys Both peers independently calculate the following:

$$\begin{aligned}
\texttt{hkey} &= \mathrm{HASH}(\texttt{secret} \,||\, \texttt{asserts}) && (23)\\
\mathit{curr}.\texttt{dkey}_{\texttt{init}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Initiator Data Key''}, 32) && (24)\\
\mathit{curr}.\texttt{dkey}_{\texttt{resp}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Responder Data Key''}, 32) && (25)\\
\mathit{curr}.\texttt{nonce}_{\texttt{init}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Initiator Nonce''}, 12) && (26)\\
\mathit{curr}.\texttt{nonce}_{\texttt{resp}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Responder Nonce''}, 12) && (27)\\
\mathit{curr}.\texttt{sess\_id}_{\texttt{init}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Initiator Session ID''}, 8) && (28)\\
\mathit{curr}.\texttt{sess\_id}_{\texttt{resp}} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Responder Session ID''}, 8) && (29)\\
\mathit{curr}.\texttt{seed}_{\infty} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Infinity Seed''}, 32) && (30)\\
\mathit{curr}.\texttt{auth} &= \mathrm{HMAC}(\texttt{hkey}, \text{``Authentication String''}, 32) && (31)\\
\texttt{chum} &= \mathrm{HMAC}(\texttt{hkey}, \text{``\$ROLE Entropy Feeder''}, 32) && (32)
\end{aligned}$$

Where $ROLE is “Initiator” or “Responder” depending on which peer is performing the calculation.

The $\texttt{seed}_{\infty}$ is how N2O implements Key Continuity[29] from ZRTP[28].

Each peer shall now ZERO the `secret` as well as the $\texttt{seed}_{\texttt{init}}$ and $\texttt{seed}_{\texttt{resp}}$, and the KEM PEER KEYS, $\texttt{peer}_{\texttt{init}}$ and $\texttt{peer}_{\texttt{resp}}$.

Last step: Each peer adds the `chum` to their respective entropy pools, then ZEROES the `chum`.

$$\text{add\_entropy}(\texttt{chum}, 32) \qquad (33)$$

Each peer needs to store the PROVENANCE in their respective peer contexts for the life of the peer context.

8.3 Network Header

N2O takes considerable effort to establish and maintain secure communications in adverse network environments. In order to efficiently receive packets, correlate the packet with correct N2O context, and decrypt the packet, we need two fields in the plaintext header: the sending peer’s `sess_id` and `nonce`.

The 64-bit (8 byte) `sess_id` is used to pair the received packet with the appropriate N2O context. The 96-bit (12 byte) `nonce` is used by the receiving peer, in conjunction with the N2O context, to decrypt the packet. Both the N2O `sess_id` and `nonce` are deterministically derived from the shared session secret. This makes it trivial to “compress” these fields on the wire.

N2O has two primary goals for the plaintext header: First, be as small as possible, and second, balance compression (size of the plaintext header fields) with the complexity of uniquely identifying the session context and detecting duplicate packets. Reliable mediums, e.g., copper traces between ICs, with one context can afford the smallest header without adding complexity. On the other hand, unreliable mediums, like ships at sea, with high packet rates and periods of lost packets require a larger header to avoid complex calculations while in the packet processing hotpath.

During N2O session context derivation, the sending `nonce` for both peers is generated. This is the first `nonce` of the session, and **must** be incremented for each packet encrypted. Incrementing the `nonce` satisfies the AEAD requirement of having a unique `nonce` for each message under the same AEAD key. Since both peers in an N2O context know the sending and receiving `nonce`s, we can truncate the `nonce` sent in the plaintext header and also use it as a per-packet counter. The per-packet counter is used to detect and drop duplicate packets.

During session context lookup, the `sess_id` should be stored in a Prefix LUT for fast and variable length lookups. In the plaintext header, the `sess_id` is *always* stored beginning with the `MSB`. This permits the receiver to do a quick prefix lookup without shifting, masking, or other onerous operations in the hotpath.

There are three forms of the plaintext header, in order of precedence: Medium, Small, and Large. The default plaintext header size is Medium. Switching between Medium and Large headers may be negotiated during N2O session negotiations. The Small header may be set by default in single-peer scenarios.

Medium Header The medium plaintext header is used when both peers have multiple peer contexts each. This header adds 8 bytes of overhead to each packet, 24 bytes total when the AEAD tag is included. This is the default header for N2O connections.

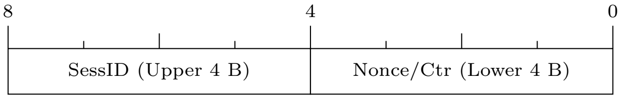

Fig. 6. N2O Medium Plaintext Header Structure

Small Header The small plaintext header is used when there is only one N2O context over a medium. For example, a serial connection between two ICs on a circuit board. This header adds 4 bytes of overhead to each packet, 20 bytes total when the AEAD tag is included.

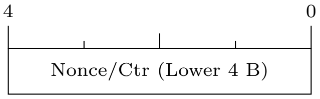

Fig. 7. N2O Small Plaintext Header Structure

Note that there is no `sess_id` in the small plaintext header. In this use case, large changes in received `nonce` may be used to detect the start of a session migration. Successful decryption with the next session's context will indicate the start of session migration.

Second note: We *must* include the counter in the header to ensure the correct `nonce` is used during decryption. The minimum plaintext header size which will preserve 32 bit alignment is 4 bytes. Therefore, we deliberately use a four byte counter when it isn't necessary so that all subsequent decrypted packet accesses are aligned in memory.

Large Header The large plaintext header is used for extended duration sessions which expect prolonged periods of packet loss in one or both directions. This header adds 16 bytes of overhead to each packet, 32 bytes total when the AEAD tag is included. If the implementor or user aren't sure if they need to use this plaintext header type, they probably don't.

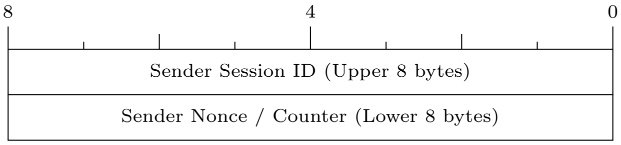
8
4
0
Sender Session ID (Upper 8 bytes)
Sender Nonce / Counter (Lower 8 bytes)

Fig. 8. N2O Large Plaintext Header Structure

Duplicate Packet Dropping The lower portion of the sending `nonce` doubles as the packet counter by which N2O detects duplicate packets and drops them. It is **critical** to the security guarantees of N2O that *all* duplicate packets be detected and dropped.

When encrypting the data portion of the packet, N2O *always* includes the plaintext header as the Additional Data in the AEAD calculation. This protects against intentional and unintentional header modification. Additionally, this ability to detect spoofed packets provides for the possibility of *lockless* duplicate packet detection, which will be described separately as an optional enhancement.

At a minimum, N2O implementations must detect and drop duplicate packets using the same algorithm Wireguard and IPSec uses, as defined in [27].

One subtle difference between IPSec[27] / Wireguard[9] and N2O is that N2O doesn't assume that a duplicate packet is necessarily a replay attack. Implementations may decide to send the identical encrypted packet over multiple network paths to probe the current connectivity state. Duplicate packets are simply a side effect of this network reconnaissance. Regardless, in all scenarios, these duplicate packets **must** be dropped.

8.4 Encrypt and Decrypt Data

We'll demonstrate the Initiator receiving data, encrypting it, and sending it to the Responder. The reverse is left as an exercise for the reader.

Encrypt Data Before the Initiator can encrypt and send data it receives, it must create the header. $\texttt{header}_{\texttt{init}}$ is one of the plaintext headers described in section 8.3. In most cases, this is the Medium length plaintext header.

N2O conceals at least 32 bits of the `nonce` on the wire to further frustrate partitioning oracle attacks[13], and other attacks described in [6].

Data sent by the Initiator is then encrypted, with $\texttt{header}_{\texttt{init}}$ included as the Additional Data:

$$\begin{aligned}\texttt{enc\_data}_{\texttt{init}} = \text{AEAD\_enc}(&\textit{i\_curr}.\texttt{dkey}_{\texttt{init}},\ \textit{i\_curr}.\texttt{nonce}_{\texttt{init}},\\ &\texttt{header}_{\texttt{init}},\ \texttt{data}_{\texttt{init}})\end{aligned} \tag{34}$$

Now, the encrypted data, including the appended authentication tag, can be appended to the header, and sent over the wire:

$$\texttt{data\_msg}_{\texttt{init}} = \texttt{header}_{\texttt{init}} \,||\, \texttt{enc\_data}_{\texttt{init}} \tag{35}$$

Last, increment the nonce. This *must* happen, regardless of success of the encryption or send steps.

$$\textit{i\_curr}.\texttt{nonce}_{\texttt{init}} + + \tag{36}$$

Decrypt Data Upon receiving the message $\texttt{data\_msg}_{\texttt{init}}$, the first step is to extract the $\texttt{sess\_id}_{\texttt{init}}$. If it matches a known peer context in the Responders' list of peers, then we proceed. Otherwise, the Responder silently drops the message. N2O uses the same packet counter tracking method as Wireguard[10]. If the specific `msg.counter` has already been received, the message *must* be silently dropped, without attempting to decrypt.

It is *vitally* important to **not** trust any header values until *after* the message has been successfully decrypted and authenticated. While N2O uses these fields to look up a peer context and detect duplicates, that is all that is done until the message authenticates. Do **not** save the reconstructed `nonce` before the message authenticates.

Assuming there is a good `sess_id` match and the `counter` is unseen, the Responder then uses the session context associated with the Initiator to reconstruct the $\texttt{nonce}_{\texttt{init}}$:

$$\texttt{nonce}_{\texttt{init}} \ = \texttt{msg.counter} \,|\, (\textit{r\_curr}.\texttt{nonce}_{\texttt{init}} \,\&\, \sim \texttt{COUNTER\_MASK}) \tag{37}$$

Do **not** set $\textit{r\_curr}.\texttt{nonce}_{\texttt{init}}$ at this time! $\texttt{nonce}_{\texttt{init}}$ is *not* trusted until the full message authenticates!

The Responder attempts to decrypt and authenticate the data:

$$\begin{aligned}\texttt{data}_{\texttt{init}} = \text{AEAD\_dec}(&\textit{r\_curr}.\texttt{dkey}_{\texttt{init}},\ \texttt{nonce}_{\texttt{init}},\\ &\texttt{header}_{\texttt{init}},\ \texttt{enc\_data}_{\texttt{init}})\end{aligned} \tag{38}$$

If, and *only* if AEAD_dec() is successful, do we continue with the following:

1. Detect and drop duplicate message via the `counter`

2. Set $r\_curr.\texttt{nonce}_{\texttt{init}} = \texttt{nonce}_{\texttt{init}}$
3. Set the $\texttt{counter\_seen}_{\texttt{init}}$ bitmask
4. Write $\texttt{data}_{\texttt{init}}$ out as appropriate
5. Increment counters in our session context with the Initiator

8.5 Session Negotiation

Session negotiations are very similar to the INTRODUCTION negotiation, except for the following changes:

1. All messages are wrapped in current session AEAD, no KEM needed
2. PROVENANCE is static, and saved into each peer's context, no need to re-send
3. $\texttt{seed}_{\infty}$ replaces the KEM PEER KEYS in the key derivation
4. $m_{\texttt{init}}$ is sent first, starting the negotiation
5. INTRODUCTION filled *curr*, all future negotiations fill *next*

A critical design feature of N2O is that *every* AEAD encrypted message contains the STATE of the sender. The included STATE serves as an integrated ACK as each peer moves from one STATE to the next. No need to worry about a lost explicit ACK, the next message to be received will confirm the state change of the sending peer. This is shown in Figure 9, below.

Dashed lines are AEAD encrypted data messages sent back and forth independently of the peer being ready to send the next message in the session negotiation. Each session negotiation message is sent, and resent on an exponential backoff, until a state change in the peer is observed. Peers only change state after receiving the next message in the session negotiation, and successfully completing the operation necessary for that message.

Data to be AEAD encrypted *never* waits for any part of the session negotiation. N2O **always** immediately encrypts and sends the data first. Session negotiation messages are attached to messages once ready, but are not required to be attached to existing messages.

Both peers begin in STATE S0_START.

To prevent nonce reuse, when an N2O process is not cleanly shutdown, it shall migrate to the *next* session context for each peer. Any peer which hasn't fully saved its *next* session shall delete that peer, and perform a fresh INTRODUCTION if desired.

Generate Fresh Seeds Exactly like Section 8.2, the peers generate fresh 32 byte, 256 bit seeds for this session:

$$\texttt{seed}_{\texttt{init}} = \text{getrandom}(32) \tag{39}$$

$$\texttt{seed}_{\texttt{resp}} = \text{getrandom}(32) \tag{40}$$

Message $m_{\texttt{init}}$ The Initiator creates and sends message $m_{\texttt{init}}$:

$$m_{\texttt{init}} = \texttt{seed}_{\texttt{init}} \tag{41}$$

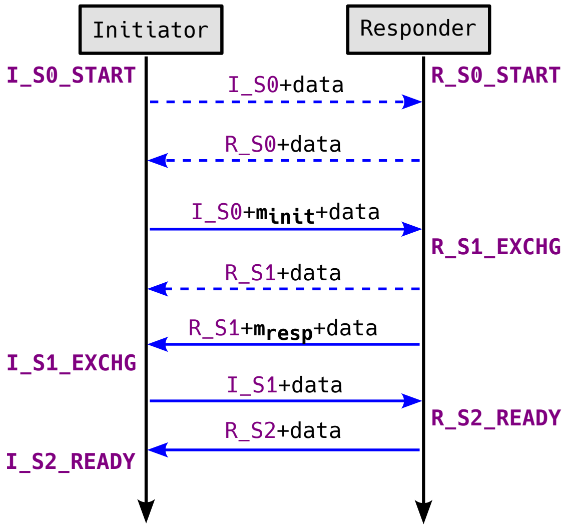

Fig. 9. N2O Session Negotiation sequence diagram

Message $m_{\texttt{resp}}$ After the Responder receives message $m_{\texttt{init}}$, the Responder creates and sends message $m_{\texttt{resp}}$:

$$m_{\texttt{resp}} = \texttt{seed}_{\texttt{resp}} \tag{42}$$

Calculate *next* Session Context Once each peer has both seeds they calculate the *next* session context, starting with the shared secret and asserts:

$$\texttt{secret} = \texttt{seed}_{\texttt{init}} \,||\, \texttt{seed}_{\texttt{resp}} \,||\, \mathit{curr}.\texttt{seed}_{\infty} \tag{43}$$

$$\texttt{asserts} = \textsc{provenance} \tag{44}$$

Unlike in the INTRODUCTION (see Section 8.2), the peers now use $\texttt{seed}_{\infty}$ calculated in Section 8.2 when performing key derivation.

Note that future sessions do **not** use the first $\texttt{seed}_{\infty}$ created during the INTRODUCTION. Rather, they use $\texttt{seed}_{\infty}$ calculated for the *curr* session context when calculating the *next* session.

With `secret` and `asserts` constructed, now each peer can independently build the *next* session context:

$$\begin{aligned}
\texttt{hkey} &= \text{HASH}(\texttt{secret} \,||\, \texttt{asserts}) && (45)\\
\mathit{next}.\texttt{dkey}_{\texttt{init}} &= \text{HMAC}(\texttt{hkey}, \text{“Initiator Data Key”},\ 32) && (46)\\
\mathit{next}.\texttt{dkey}_{\texttt{resp}} &= \text{HMAC}(\texttt{hkey}, \text{“Responder Data Key”}, 32) && (47)\\
\mathit{next}.\texttt{nonce}_{\texttt{init}} &= \text{HMAC}(\texttt{hkey}, \text{“Initiator Nonce”},\ 12) && (48)\\
\mathit{next}.\texttt{nonce}_{\texttt{resp}} &= \text{HMAC}(\texttt{hkey}, \text{“Responder Nonce”}, 12) && (49)\\
\mathit{curr}.\texttt{sess\_id}_{\texttt{init}} &= \text{HMAC}(\texttt{hkey}, \text{“Initiator Session ID”}, 8) && (50)\\
\mathit{curr}.\texttt{sess\_id}_{\texttt{resp}} &= \text{HMAC}(\texttt{hkey}, \text{“Responder Session ID”}, 8) && (51)\\
\mathit{next}.\texttt{seed}_{\infty} &= \text{HMAC}(\texttt{hkey}, \text{“Infinity Seed”}, 32) && (52)\\
\mathit{next}.\texttt{auth} &= \text{HMAC}(\texttt{hkey}, \text{“Authentication String”}, 32) && (53)\\
\texttt{chum} &= \text{HMAC}(\texttt{hkey}, \text{“\$ROLE Entropy Feeder”}, 32) && (54)
\end{aligned}$$

Where $ROLE is “Initiator” or “Responder” depending on which peer is performing the calculation. Each peer adds the `chum` to their respective entropy pools, then ZEROES the `chum`.

$$\text{add\_entropy}(\texttt{chum},\ 32) \tag{55}$$

Once each peer has completed calculating the *next* session context, they immediately change their respective STATE to `S1_EXCHG`.

After each peer receives a message with peer STATE `S1_EXCHG`, they add their peers *next*.`sess_id` as a valid pointer to the peer context. This enables them to automatically detect session changes initiated by the peer without losing any data. Once the *next*.`sess_id` is recorded, the peers migrate themselves to the final STATE, `S2_READY`. This indicates that the peer is ready to migrate to the next session when it chooses to do so, or when it receives a message from its peer encrypted with the *next* session context.

Once both peers reach the STATE `S2_READY`, either peer can force a session migration simply by beginning to use the *next* session context. See Section 8.6 for details.

8.6 Session Migration

Session migration is seamless, N2O never blocks, throttles, or limits the transmission of ordinary data. When a peer decides it's time to transition to the *next* session context, it simply does so. Everything necessary has been prepared ahead of time. Either peer can initiate a session migration, regardless of role.

When to Migrate Sessions N2O adds no constraints on top of the constraints provided by the AEAD systems used. N2O recommends rotating sessions daily or weekly, as there is zero cost and it only strengthens the shared context by adding more entropy.

Since the `sess_id` and initial `nonce` are assigned per-peer, per-session, deployments may prefer to rotate sessions in sync with changes in network routes. Doing so would reduce the ability to correlate multiple network connections to the same peer pair.

Migration State Machine Technically, there is a state machine. However, no separate variable is necessary to track the state. Regardless of which peer initiates the migration, both peers *must* retain the *curr* session, and decrypt messages with it, until they receive a message from the other peer encrypted with the *next* session keys.

At that time, the peer sets its STATE to `SO_START`, and can wipe *curr*. At a low level, the peer may do an atomic pointer swap of *next* and *curr*.

Session negotiation for the new *next* session can now begin.

9 Conclusion

N2O is a robust, modern, post-quantum encrypted tunnel. It prioritizes simplicity, data transfer, and unobtrusiveness above all else. N2O learns lessons from the great projects that have come before it.

10 Thanks

The author is deeply indebted to friends and family who tolerated his reclusiveness while N2O was committed to paper.

Jason would like to thank Vinnie Moscaritolo for reviewing very rough versions of this document, and for the lifetime of expertise he brought to bear during his reviews.

Jason thanks Jon Callas for his careful review comments and deep insight from decades of creating secure, useful cryptographic primitives and protocols.

Jason thanks Jack O'Connor for his assistance with how to securely perform the N2O key derivations with Blake3.

References

1. Agency, U.S.N.S.: Commercial national security algorithm suite 2.0. [Online; accessed 04-September-2024], https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF
2. et al, E.R.: The transport layer security (tls) protocol version 1.3. [Online; accessed 04-September-2024], https://datatracker.ietf.org/doc/html/rfc8446
3. et al, J.Y.: Openvpn on github. [Online; accessed 04-September-2024], https://github.com/OpenVPN/openvpn
4. Arciszewski, S.: Announcing aes-gem (aes with galois extended mode). [Online; accessed 05-September-2024], https://blog.trailofbits.com/2024/07/12/announcing-aes-gem-aes-with-galois-extended-mode/
5. Bragin, M., Santos, M., UG, W.: Netbird. [Online; accessed 05-September-2024], https://github.com/netbirdio/netbird, see https://docs.netbird.io
6. Degabriele, J.P., Govinden, J., Günther, F., Paterson, K.G.: The security of chacha20-poly1305 in the multi-user setting. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security **CCS 21** (2021)
7. Dierks, T., et al, E.R.: The transport layer security (tls) protocol version 1.2. [Online; accessed 04-September-2024], https://datatracker.ietf.org/doc/html/rfc5246
8. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **IT-22**(6), 644–654 (1976)
9. Donenfeld, J.A.: Wireguard: Next generation kernel network tunnel. [Online; accessed 04-September-2024], https://www.wireguard.com/papers/wireguard.pdf
10. Donenfeld, J.A.: Wireguard: Next generation kernel network tunnel. [Online; accessed 04-September-2024], https://www.wireguard.com/papers/wireguard.pdf, section 5.4.6: Transport Data Messages, last paragraph
11. Dworkin, M.: Nist sp 800-38d: Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. [Online; accessed 04-September-2024], https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf
12. Kerkour, S.: Chacha20-blake3. [Online; accessed 05-September-2024], https://kerkour.com/chacha20-blake3
13. Len, J., Grubbs, P., Ristenpart, T.: Partitioning oracle attacks. Cryptology ePrint Archive, Paper 2020/1491 (2020), https://eprint.iacr.org/2020/1491
14. Nir, Y., Langley, A.: Chacha20 and poly1305 for ietf protocols. [Online; accessed 04-September-2024], https://datatracker.ietf.org/doc/html/rfc7539
15. O'Connor, J., Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z.: Blake3: One function, fast everywhere. [Online; accessed 04-September-2024], https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf
16. Roberto Avanzi, Joppe Bos, L.D.E.K.T.L.V.L.J.M.S.P.S.G.S., Stehlé, D.: Crystals-kyber (version 3.02). [Online; accessed 04-September-2024], https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf
17. Roberto Avanzi, Joppe Bos, L.D.E.K.T.L.V.L.J.M.S.P.S.G.S., Stehlé, D.: Crystals-kyber (version 3.02). [Online; accessed 04-September-2024], https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf, section 1.4: Kyber Parameter Sets
18. of Standards, N.I., Technology: Fips 197: Advanced encryption standard (aes). [Online; accessed 04-September-2024], https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf

19. of Standards, N.I., Technology: Fips pub 180-3: Secure hash standards (shs). [Online; accessed 04-September-2024], https://csrc.nist.gov/files/pubs/fips/180-3/final/docs/fips180-3_final.pdf
20. of Standards, N.I., Technology: Fips pub 198-1: The keyed-hash message authentication code (hmac). [Online; accessed 04-September-2024], https://doi.org/10.6028/NIST.FIPS.198-1
21. of Standards, N.I., Technology: Module-lattice-based key-encapsulation mechanism standard. [Online; accessed 04-September-2024], https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf
22. of Standards, N.I., Technology: Module-lattice-based key-encapsulation mechanism standard. [Online; accessed 04-September-2024], https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf, section 8: Parameter Sets
23. of Standards, N.I., Technology: Module-lattice-based key-encapsulation mechanism standard. [Online; accessed 04-September-2024], https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf, appendix C: Differences From the CRYSTALS-Kyber Submission
24. of Standards, N.I., Technology: Nist post-quantum competition. [Online; accessed 04-September-2024], https://www.nist.gov/pqcrypto
25. Tailscale: Tailscale. [Online; accessed 05-September-2024], https://github.com/tailscale/tailscale, see https://tailscale.com/kb/1376/tech-overviews as well
26. Ylonen, T.: Openssh. [Online; accessed 05-September-2024], https://www.openssh.com/specs.html, rFCs 4250, 4251, 4252c, 4253c, 4254c
27. Zhang, X., Zou), T.T.T.: Ipsec anti-replay algorithm without bit shifting. [Online; accessed 29-MAR-2025], https://datatracker.ietf.org/doc/html/rfc6479
28. Zimmermann, P., Callas, J., Johnston, A.: Zrtp: Media path key agreement for unicast secure rtp. [Online; accessed 03-September-2024], https://datatracker.ietf.org/doc/html/rfc6189
29. Zimmermann, P., Callas, J., Johnston, A.: Zrtp: Media path key agreement for unicast secure rtp. [Online; accessed 03-September-2024], https://datatracker.ietf.org/doc/html/rfc6189, section 15.1: Self-Healing Key Continuity Feature
30. Zimmermann, P., Callas, J., Johnston, A.: Zrtp: Media path key agreement for unicast secure rtp. [Online; accessed 03-September-2024], https://datatracker.ietf.org/doc/html/rfc6189, section 7: Short Authentication String