TECHNICAL DESCRIPTION OF THE ALGORITHM

“Method of dynamic generation of identifiers (Rolling ID) for Meshtastic networks”

2026 г.

CONTENTS

Description of the operation of the Rolling ID algorithm

1. General description

The **Rolling ID** algorithm is designed to provide routing anonymity in mesh networks based on LoRa technology (in particular, the Meshtastic protocol). The method replaces the static global node identifier (NodeID) with a dynamically computed pseudonym that is periodically changed based on time and a shared secret key.

Goal: To prevent the collection of metadata, the construction of a social graph, and the physical tracking of users by passive observers.

2. Variables and constants

The algorithm uses the following variables and parameters:

Symbol	Name	Data type	Description
K	Pre-Shared Key (PSK)	Byte Array (256 bit)	Shared channel secret key (Channel Name/PSK). Known to all trusted members of the group.
t	Timestamp	Integer (Uint64)	Current time in UNIX timestamp format (seconds since 01/01/1970).
T_{epoch}	Epoch Duration	Integer	Constant for the lifetime of a single ID. **Recommended value:** 900

			seconds (15 minutes).
C	Current Epoch	Integer (Uint32)	Time interval counter (epoch). Used as a 'salt' for hashing.
H	Hash Output	Byte Array (256 bit)	The complete output of a cryptographic hash function.
$ID_{rolling}$	Rolling NodeID	Uint32 (32 bit)	Final dynamic identifier written into the LoRa packet header.
W	Window Size	Integer	Width of the verification window to compensate for clock desynchronization. **Recommended value:** W=1
N	Contacts Count	Integer	Number of trusted contacts in the recipient node's local database.
S_{valid}	Valid Set	Set of Uint32	The set of valid identifiers for a specific contact at a given point in time.

3. Mathematical model

3.1. Time discretization

To compensate for small clock offsets and transmission delays, continuous time t is converted into discrete epochs C.

$$C = \lfloor \frac{t}{T_{epoch}} \rfloor$$

Where the operation ⌊x⌋ denotes rounding down to the nearest integer.

3.2. Generation of a pseudorandom sequence

The generator uses the HMAC (Hash-based Message Authentication Code) mechanism with the SHA-256 function. This ensures irreversibility (the impossibility of computing K from a given ID) and an avalanche effect.

$$H = HMAC_{SHA256}(K, C)$$

The inputs are the secret key K and the current epoch number C (represented in byte form).

3.3. Truncation

Since the Source NodeID field in the Meshtastic header is limited to 4 bytes (32 bits), the 256-bit hash H is truncated. The most significant 4 bytes (MSB — Most Significant Bytes) are taken.

Final generation formula:

$$ID_{rolling} = MSB32\left(HMAC_{SHA256}\left(K, \lfloor \frac{t}{900} \rfloor\right)\right)$$

4. Operating algorithms

4.1. Sending algorithm (Sender Side)

Executed before forming each batch or when the epoch changes.

1. Get the current system time t
2. Compute the current epoch $C = \lfloor \frac{t}{T_{epoch}} \rfloor$.
3. If *C* has changed compared to the previous transmission or the packet is being sent for the first time:
   - Calculate using the formula from section 3.3.
   - Update the "Current NodeID" variable in the network stack.
4. Form a LoRa packet by writing the obtained $ID_{rolling}$ into the Unencrypted Header in the Source field.
5. Transmit a packet

4.2. Reception and validation algorithm (Receiver Side)

Executed when any over-the-air packet with an unknown Source ID is received. The "Sliding Verification Window" method is used

Input: Packet with an identifier $ID_{received}$

Settings: Local contacts database $\{K_1, K_2, ..., K_n\}$.

1. 1. Get the current time t and calculate the local epoch C_{local}
2. 2. Set the range of epochs to check: $[C_{local} - W,\ C_{local} + W]$.
   When $W = 1$, the epochs $\{C - 1,\ C,\ C + 1\}$ are checked.
3. Loop over all contacts i from 1 to N :
   – Take the contact's key K_i.
   – Loop over window offsets $j \in \{-W, ..., +W\}$:
     ▪ Compute the candidate $ID_{candidate} = GenerateID(K_i, C_{local} + j)$.
     ▪ **Check:** if $ID_{received} == ID_{candidate}$:
       - The packet is considered valid.
       - The sender (contact i) is identified.

- The packet is passed to the AES-256 decryption layer.
- **End of the algorithm (Success).**

4. If no matches are found after checking all contacts and windows, the packet is ignored.

4.3. Procedure "Soft Handshake"

To maintain network connectivity when the epoch changes, proactive route updates are used.

1. **Event:** a new epoch has begun $\left(t \bmod T_{epoch} == 0\right)$
2. The node computes its ID_{new}
3. The node sends a broadcast Keep-Alive packet (or an empty telemetry packet) signed with ID_{new}
4. Neighboring trusted nodes accept the packet and validate it using Algorithm 4.2 (since the window $W = 1$ allows accepting a "future" epoch, or if the clocks are desynchronized).
5. Trusted nodes update the entry in the Dynamic Mapping Table (see item 5).

5. Integration into the network layer

To ensure compatibility with the existing architecture, **Dual-Layer Addressing** is introduced.

Dynamic Lookup Table

Stored in the RAM of each trusted node.

Field	Description

Static ID	An immutable internal identifier (for example, !f38a20). Used for display in the chat and for the user to select the recipient. Never transmitted in plain text.
Current Rolling ID	Current external alias (for example, 0x4A1B2C3D). Used for routing at the LoRa level.
Last Seen Epoch	Timestamp of the last activity (for cleaning up stale records).

Routing logic:

When sending a message to User 1 (!b224aa):

- Find !b224aa in the table.
- Extract the corresponding Current Rolling ID.
- Specify this ID in the Destination field of the packet header.

6. Estimation of collision probability

The probability that two different nodes in the same group will choose the same 32-bit ID in the same epoch (the birthday paradox).

$$P(collision) \approx 1 - e^{-\frac{N^2}{2M}}$$

Where:

- $M = 2^{32}$ (size of the address space $\approx 4.29 \times 10^9$).
- N - number of active nodes

Calculated values:

- For 2 nodes $P \approx 2.3 \times 10^{-10}$
- For 100 nodes $P \approx 1.16 \times 10^{-6}$

7. Resource intensity (Performance Metrics)

- **Network overhead:** 0 bytes (existing header fields are used).
- **Computational complexity:**
  - **Sender:** Hashing once every 15 minutes.
  - **Receiver:** $3 \times N$ hashing for each incoming packet.
  - For $N = 50$ contacts ≈ 150 HMAC-SHA256 operations. On an ESP32 (240 MHz) this takes less than 5–10 ms.
- **Energy consumption:** negligible impact, comparable to normal packet processing.

8. Software model of the algorithm

```
import hmac
import hashlib
import time
import struct

# --- ALGORITHM CONSTANTS ---
T_EPOCH = 900          # Epoch duration: 15 minutes (900 seconds)
WINDOW_SIZE = 1        # Validation window: +/- 1 epoch
PSK = b"secret_key_256_bit_length_shared"  # Pre-shared key (Shared secret)

def generate_rolling_id(shared_key, timestamp):
    """
    Generates a dynamic Rolling ID.
    Implements the formula: ID = MSB32(HMAC-SHA256(K, floor(t / T_epoch)))
    """
    # 1. Time discretization (Determine the current epoch)
    current_epoch = int(timestamp // T_EPOCH)

    # Convert epoch number to byte format (4 bytes, big-endian)
    epoch_bytes = struct.pack(">I", current_epoch)
```

```
    # 2. Compute HMAC-SHA256
    hash_full = hmac.new(shared_key, epoch_bytes, hashlib.sha256).digest()

    # 3. Truncation - take the first 4 bytes (32 bits)
    # And interpret them as an integer (uint32)
    rolling_id = struct.unpack(">I", hash_full[:4])[0]

    return rolling_id

def verify_rolling_id(shared_key, received_id, current_timestamp):
    """
    Receiver-side validation algorithm.
    Uses a Sliding Verification Window method.
    """
    local_epoch = int(current_timestamp // T_EPOCH)

    # Check candidates within the range [C-W, C+W]
    for offset in range(-WINDOW_SIZE, WINDOW_SIZE + 1):
        check_epoch = local_epoch + offset
        epoch_bytes = struct.pack(">I", check_epoch)

        # Generate the expected ID for this specific epoch
        hash_check = hmac.new(shared_key, epoch_bytes, hashlib.sha256).digest()
        candidate_id = struct.unpack(">I", hash_check[:4])[0]

        if received_id == candidate_id:
            return True, offset  # ID is valid; return offset for clock synchronization

    return False, 0  # ID not found among trusted candidates

# --- USAGE EXAMPLE ---
if __name__ == "__main__":
    now = int(time.time())

    # Node A (Sender) generates its current ID
    my_id = generate_rolling_id(PSK, now)
    print(f"Sender: Generated Rolling ID: {hex(my_id)}")

    # Node B (Receiver) receives the packet and verifies it
    # Suppose Node B's clock is 2 minutes (120 seconds) ahead
    time_on_receiver = now + 120

    is_valid, drift = verify_rolling_id(PSK, my_id, time_on_receiver)

    if is_valid:
```

```
    print(f"Receiver: Packet is valid. (Epoch drift: {drift})")
else:
    print("Receiver: Validation failed. Packet ignored.")
```

Software model of the Rolling ID algorithm in Python.

9. License information