OSNMAlib: An Open Python Library for Galileo OSNMA

Aleix Galan
Dept. of Telecommunications
UAB
Barcelona, Spain
ORCID: 0000-0002-5762-6982

Ignacio Fernandez-Hernandez
DG DEFIS
European Commission
Brussels, Belgium
ORCID: 0000-0002-9308-1668

Luca Cucchi
Joint Research Center
European Commission
Ispra, Italy
luca.cucchi@ec.europa.eu

Gonzalo Seco-Granados
IEEC-CERES
UAB
Barcelona, Spain
ORCID: 0000-0003-2494-6872

***Abstract*—Galileo has started authenticating its navigation message through OSNMA. In order to support OSNMA implementation by receiver manufacturers and application developers, this paper presents OSNMAlib, an open Python library implementing OSNMA functions. OSNMAlib processes the Galileo I/NAV pages in decoded SBF, hexadecimal or other formats, and performs the required operations to authenticate Galileo navigation data: OSNMA status handling, cryptographic functions required for the Merkle tree, digital signatures, keychain management and tag authentication. It handles the up-to-date data authentication status and performs the public key and chain renewal and revocation processes. This paper describes OSNMAlib architecture and its main functions and presents the first test results.**

***Index Terms*—Galileo, GNSS, OSNMA, Authentication, Open Implementation, Open Source, Python, OSNMAlib**

I. Introduction

Galileo has started broadcasting its Open Service Navigation Message Authentication feature, or OSNMA [1]. OSNMA aims at authenticating the Galileo navigation message through cryptographic functions. By authenticating the navigation message, spoofing attacks to Galileo, and GNSS applications in general, become much more difficult, as nowadays all civil signals are completely unauthenticated. In addition to data authentication, OSNMA makes its carrier signal, Galileo E1-B, more difficult to replay, allowing the protection of receivers against replay attacks [2] [3].

Galileo OSNMA requires the receiver to implement the necessary cryptographic functions to authenticate the data, which is its ultimate purpose [4]. It also requires the receiver to have a loosely time reference, between some seconds to some minutes, depending on the use case, in order to initalise securely the OSNMA protocol. The receiver logic and cryptographic functions are not trivial to implement for receiver manufacturers and application developers whose focus is in the location performance.

With the purpose to facilitate the implementation of OSNMA, *OSNMAlib* is an open source Python library that can be integrated in existing receivers and applications to incorporate navigation message authentication to the positioning process. It can read the Galileo I/NAV pages when received, store the navigation and authentication data, perform the authentication verification, and report the status. The platform also allows to optimize the OSNMA processing in order to reduce time to authenticated fix, and improve availability, among other improvements. The *OSNMAlib* implementation can be found in the following repository [5]. While not open-source to the knowledge of the authors, other OSNMA reported implementations include [6], [7] and [8].

The next section presents the overall *OSNMAlib* architecture. Then, the paper presents the receiver main functions and a user execution guide. The next sections cover some specifics of the library: data identification, TESLA key management, and subframe generation. After that, the paper presents some *OSNMAlib* test results, and finalises with the conclusions.

II. OSNMAlib Architecture

This section reviews how the *OSNMAlib* receiver works at high level, and its start procedure. This approach will allow obtaining a general understanding of the receiver without the need for coding knowledge. However, previous knowledge of Galileo OSNMA protocol is strongly advised. At the time of writing, the latest OSNMA full specification published is provided in [9].

A. Simplified Internal Architecture

The simplified internal architecture of the *OSNMAlib* receiver comprises the necessary core classes to understand the workflow of the software. The relationship of these classes is exemplified using a class diagram (Fig. 1). For each class, we provide a functionality definition, and the definition order follows the logic path of the data entering the receiver.

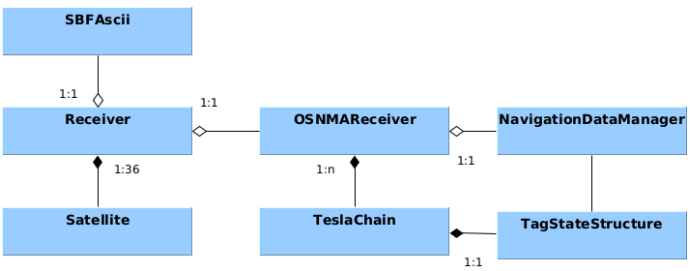

Fig. 1. OSNMA receiver simplified class diagram.

978-1-6654-1616-0/22/$31.00 ©2022 IEEE

- *Receiver*: Module that emulates a Galileo receiver. It is responsible for reading the navigation message, rely each page to the *OSNMAReceiver* module to extract the ADKD data, and reconstruct the OSNMA subframe messages. To do so, it tracks the satellites in view and stores the 30-second subframe that constitutes the OSNMA data messages using the class *Satellite*, which handles subframe synchronism and completeness. The receiver module is also in charge of checking the time synchronization at the start of the execution and verifying that the pages are suitable to be used (no alert pages, have the correct CRC and belong to the E1B component).
- *SBFAscii*: Input module for the *Receiver* class. The SBFAscii class is an iterator that reads the navigation message from a CSV file and encapsulates it in a *DataFormat* object. The *DataFormat* object contains the 240 bits of the Galileo I/NAV double page transmitted, the SVID of the satellite transmitting it, and the transmission GST at the start of the page.
- *Satellite*: Class that stores the OSNMA data in each page to reconstruct the OSNMA subframe messages. It checks if the satellite is transmitting OSNMA information and tracks which pages are received for every subframe. In the HKROOT case, different pages from satellites transmitting the same block can be merged.
- *OSNMAReceiver*: Module that handles all the OSNMA protocol-related actions. The *OSNMAReceiver* class maintains the protocol state: which Public Key is in use, which is the TESLA Chain in force, the last authenticated NMA Header, etc. It is responsible of verifying the HKROOT messages and start any related action: creating a *TESLAChain* object, storing a new Public Key, updating the internal state to the appropriate special event, etc. With the information of the TESLA chain in force, the *OSNMAReceiver* module also parses and stores the MACK message into the *TESLAChain* object.
- *NavigationDataManager*: The *NavigationDataManager* module stores the navigation data by ADKD. It allows to retrieve the navigation data linked to an authentication Tag using the GST of when the Tag was received and not the IOD value. The module also stores the authenticated Tags for the same navigation data set, a navigation data set is authenticated after accumulating a user defined number of Tag bits.
- *TESLAChain*: This module represents the TESLA Chain and is generated with the information retrieved from a verified KROOT. It provides the necessary methods to verify a received TESLA Key in an efficient way, and updates the stored KROOT with every new floating key transmitted. Only one TESLA Chain object is in force in any given moment, and for each one a *TagStateStructure* object is created. The *TESLAChain* retrieves the Tags and TESLA Key/s from the MACK messages, verifies the TESLA Key/s and relies the tags to the *TagStateStructure*.
- *TagStateStructure*: The *TagStateStructure* class performs all the necessary operations for the correct tag authentication. Firstly, it checks that the Tags in the MACK message follow the MAC Lookup Table structure. For the *FLX* Tags, it waits until being able to compute the *MACSEQ* verification. The correctly structured Tags are stored in another list waiting for their TESLA Key. When that TESLA Key is retrieved, the receiver checks for each Tag if they have already received the data they authenticate. If a Tag has not received the appropriate navigation data before the first bit of it's TESLA Key was propagated, the Tag is discarded. The Tags with valid data are authenticated.

B. Start Sequence

When the OSNMAlib receiver is executed, it performs certain steps before reaching the *Started* state. The receiver is considered to be in *Started* state when it has authenticated the DSM-KROOT message for the TESLA Chain in force.

The receiver starts assuming that the NMA Status is not *Don't use* and the CPKS is *Nominal*. The reason behind this decision is that the receiver cannot trust the retrieved NMA Status and CPKS until they have been authenticated; otherwise there would be a thread vector for denial of service vulnerabilities. After authenticating a DSM-KROOT message, the receiver handles the NMA Status and CPKS according to the specifications.

The receiver starts in the *Cold Start* state, without Public Key or KROOT. The first thing it does is try to retrieve a Public Key from a specified file or from the GSC website. If it cannot retrieve a Public Key from those channels, the receiver will wait until a DSM-PKR message is transmitted and the Public Key received is authenticated.

When receiver has a verified Public Key (from any channel) but no KROOT yet, it is in *Warm Start* state. Then, the receiver waits until retrieving a DSM-KROOT from the navigation message. When retrieving a DSM-KROOT, it checks with the Public Key ID that the Public Key needed to authenticate the Digital Signature of the KROOT is the same as the one stored. If the value is different, it discards the Public Key and retrieves another one. If the Public Key IDs are the same, then it proceeds to authenticate the KROOT. If the authentication fails, the Public Key is discarded and another one retrieved. If the authentication is successful, the KROOT is verified and its parameters can be interpreted. If the KROOT verified corresponds to the TESLA chain in force, as it may be read during an EOC event and be the KROOT for the next TESLA chain, the receiver is considered *Started*.

There are only 2 valid cases where the receiver can read a KROOT that is not in force: in the Step 1 of the End of Chain event (EOC) and in the Step 1 of the Public Key Revocation event (PKREV). In both cases, the key read will be the next one to be in force, so the receiver stores the key and retrieves another one. In the case of EOC, this will be repeated until the KROOT in force is retrieved or the EOC event ends. In the case of PKREV, the iteration will continue until the PKREV changes from Step 1 to Step 2.

If the receiver has a verified Public Key and a stored KROOT at the start of the execution, the receiver is set to *Hot Start* state. In this case, the receiver will wait until authenticating a TESLA Key with the stored KROOT. If the authentication is successful, the receiver is *Started*. Otherwise, the stored KROOT is discarded.

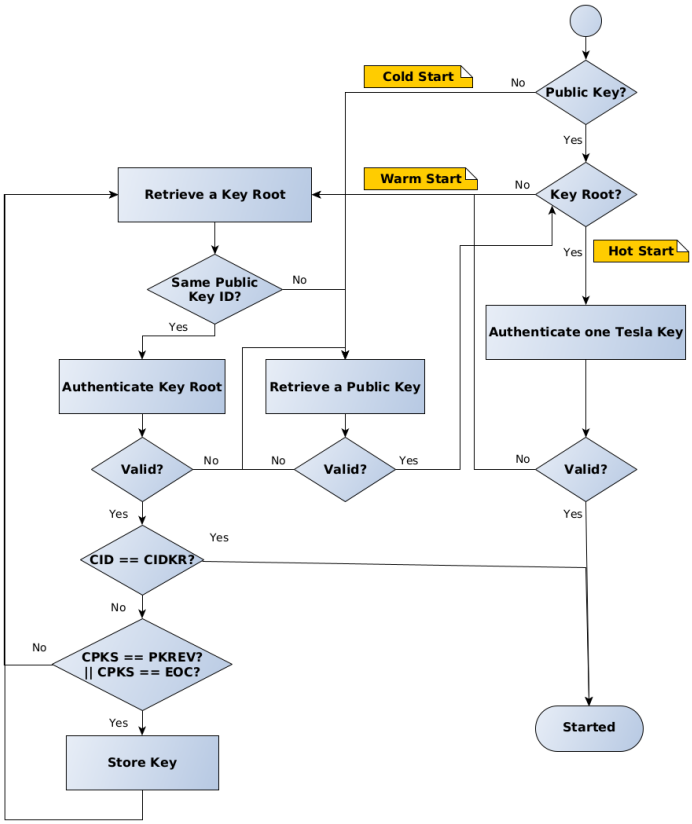

Fig. 2. OSNMA receiver start flowchart considering the stored cryptography data.

III. Receiver Functionality

This section is intended as a quick guide to run the OSNMA Open Implementation Receiver software. It presents the execution of test scenarios with real broadcasted OSNMA data and the execution of the software with custom Galileo data.

The user may be required to perform basic Python scripting when running the receiver with custom data, although the software is provided with templates and comments to minimize the hassle. The OSNMA Open Implementation Receiver software is intended to run on Python versions 3.7, 3.8, and 3.9.

A. Test Execution

The software is provided with several test scenarios under the folder *onsnma/tests/scenarios/*. The scenarios cover different configurations and events of the OSNMA protocol. The data used by these tests was recorded during the OSNMA Internal Test Phase (November, 2020 - April, 2021).

The tests can be run by executing the file *receiver_test.py* under *osnma/test/* folder. Keep in mind that this execution may take a while, since each test comprises several hours of satellite data.

By default, all tests are executed with *info* logging level on the file handler. That is, the log files will contain the maximum amount of information. This log files are stored under the folder *osnma/tests/test_logs*. For each sub-test (in this case, for each scenario) a subfolder is created with the name format *logs_YYYYmmdd_HHMMSS*. Finally, the log files are stored inside their respective subfolder.

To run the tests it is recommended to use the Python framework *pytest* [10], although they can be run calling the traditional Python interpreter.

1) Pytest: The *pytest* framework is installed along the requirements for the software and is the easiest way to execute the OSNMA Open Implementation receiver tests. To do so, the following shell commands are provided. Note that the users interpreter work directory is assumed to be the top folder of the provided software and *python pip* shall be already installed.

Listing 1. Test execution with Pytest

```
$ pip install -r requirements.txt
$ cd tests
$ pytest receiver_test.py
```

2) Python interpreter: The tests can also be executed using the traditional Python interpreter. In that case, the following shell commands should be executed.

Listing 2. Test execution with Python interpreter

```
$ pip install -r requirements.txt
$ cd tests
$ python3 receiver_test.py
```

B. Nominal Scenarios

During the Internal Test Phase, only six TESLA Chain configurations where transmitted. For each of them, a test scenario of around one hour of navigation data has been developed. These six configuration scenarios are nominal executions: there is no special event taking place (TESLA Chain renewal/revocation, public key renewal/revocation, etc).

The test scenarios for each configuration in nominal transmission are stored under the folder *osnma/tests/scenarios/config[configuration_number]*. The Public Key in use during each scenario is already stored in the folder. Table I shows the parameter selection for Configuration 1 and all the possible parameters that change between configurations.

C. Special Events

Several special events are also included in the test scenarios. The data set available for the project did not cover the TESLA Chain revocation and OSNMA Alert Message events. Although the receiver has been programmed to handle these events, they have not been tested with real satellite data.

TABLE I
OSNMA PARAMETERS FOR CONFIGURATION 1.

Parameters	Settings
Tag size	40 bits
Key size	128 bits
Number of MACK blocks per sub-frame (NMACK)	1
Number of tags per MACK block (n_t)	6
Digital Signature algorithm	ECDSA P-256
Nb. of blocks in the DSM-HKROOT (NB_{DK})	8
Hash Function	SHA-256
MAC Function	HMAC-SHA-256
MACLT	27

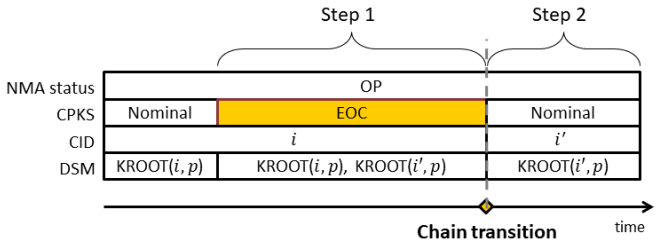

Fig. 3. End of Chain event.

1) End Of Chain - Step 1: This test scenario comprises one hour from the Step 1 of the End Of Chain event. It is stored under the folder *osnma/tests/scenarios/eoc*, the Public Key in use is identified with PKID 9 and is already stored in the folder. The configuration in force is the number 6 transitioning to the number 1.

The data available during the test phase had some minor differences with respect to the applicable ICD. The receiver handles the EOC event correctly, and will still work with the ICD consistent data. However, there will be more efficient ways of handling the EOC event once the data is updated.

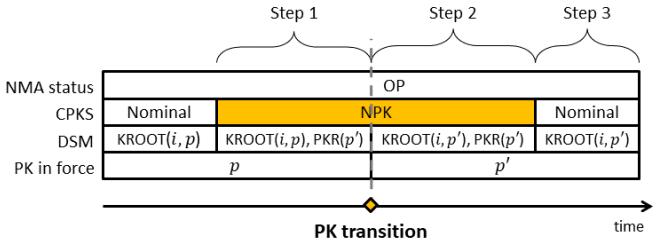

Fig. 4. New Public Key event.

2) New Public Key - Step 1: This test scenario comprises one hour from the Step 1 of the New Public Key event. It is stored under the folder *osnma/tests/scenarios/npk_1*, the Public Key in use is identified with PKID 3 and is already stored in the folder. The next Public Key is identified with PKID 4.

3) New Public Key - Step 2: This test scenario comprises one hour from the Step 2 of the New Public Key event. It is stored under the folder *osnma/tests/scenarios/npk_2*, the Public Key in use is identified with PKID 4 and is already stored in the folder.

4) New Public Key - Step 1 and 2: This test scenario comprises one hour from the Step 1 of the New Public Key event and 1 hour from the Step 2, to simulate the transition instant. This generates a GST gap between both data files, but the receiver can handled it correctly. It is stored under the folder *osnma/tests/scenarios/npk_12*, the Public Key in use at the start of the scenario is identified with PKID 3 and is already stored in the folder. The next Public Key is identified with PKID 4.

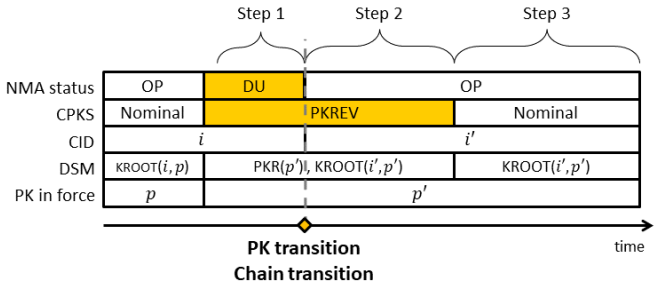

Fig. 5. Public Key Revocation event.

5) Public Key Revocation - Step 1: This test scenario comprises one hour from the transition from Nominal to Step 1 of the New Public Key event. It is stored under the folder *osnma/tests/scenarios/pkrev_1*, the Public Key in use is identified with PKID 4 and is already stored in the folder. The next Public Key is identified with PKID 5.

6) Public Key Revocation - Step 2: This test scenario comprises one hour from the transition from Step 1 to Step 2 of the New Public Key event. It is stored under the folder *osnma/tests/scenarios/pkrev_2*, the Public Key in use is not stored in the folder. The Public Key is identified with PKID 5.

7) Public Key Revocation - Step 2 with Public Key: This test scenario comprises one hour from the transition from Step 1 to Step 2 of the New Public Key event. It is stored under the folder *osnma/tests/scenarios/pkrev_2_with_pk*, the Public Key in use is identified with PKID 5 and is already stored in the folder.

8) Public Key Revocation - Step 1 and 2: This test scenario comprises one hour from the transition from Nominal to Step 1 of the New Public Key event and 1 hour from the transition from Step1 to Step 2, to simulate the complete event flow. This generates a GST gap between both data files, but the receiver can handled it correctly. It is stored under the folder *osnma/tests/scenarios/pkrev_12*, the Public Key in use at the start of the scenario is identified with PKID 4 and is already stored in the folder. The next Public Key is identified with PKID 5.

D. Start scenarios

These test scenarios comprise different possible outcomes of the receiver start sequence. They are tested on the con-

figuration 1 nominal transmission. Some cases are not tested because they are already handled in some of the previous tests.

1) Correct Hot Start: Test scenario for the configuration 1 in nominal transmission with the Key Root for the TESLA Chain in force already stored. It is stored under the folder *osnma/tests/scenarios/start_scenarios/config1_hot_start*. The Public Key in use during the scenario is identified with PKID 1 and is already stored in the folder.

The receiver will verify the stored KROOT with the current Public Key. Then, after verifying correctly a TESLA Key, it will consider that the start process has finished satisfactorily.

2) Wrong KROOT on Hot Start: Test scenario for the configuration 1 in nominal transmission with the Key Root for a the TESLA Chain that is not the one in force already stored. It is stored under the folder *osnma/tests/scenarios/start_scenarios/config1_wrong_kroot*. The Public Key in use during the scenario is identified with PKID 1 and is already stored in the folder.

The receiver will verify the stored KROOT with the current Public Key. The verification will fail and the receiver will fallback to Warm Start (having only the Public Key).

3) Wrong Public Key on Warm Start: Test scenario for the configuration 1 with a wrong Public Key provided. It is stored under the folder *osnma/tests/scenarios/start_scenarios/config1_wrong_kroot*. The wrong Public Key is identified with PKID 3 and is already stored in the folder. They Public Key in force is identified with PKID 1.

The receiver will load the Public Key stored and set itself on a Warm Start. After reading a DSM KROOT, it will try to verify the read message with the Public Key loaded. The verification will fail because the PKID do not match, therefore the receiver will discard the Public Key and fallback to a Cold Start.

IV. Execution with Custom Data

The OSNMA Open Implementation receiver can be used with custom data files. However, the receiver is only guaranteed to work with data consistent with the OSNMA User ICD for the Test Phase version 1.0.

A. Data Format

The receiver works by instantiating an iterator that, for each iteration, returns the iteration index and a *DataFormat* object. The *DataFormat* class and the different iterators are defined under the Python file *osnma/receiver/input.py*.

In Python, an iterator is an object which implements the iterator protocol, which consist of the methods *__iter__()* and *__next__()*. The *__iter__()* method allows to do some initializing, but must always return the iterator object itself. The *__next__()* method also allows to do operations, and must return the next item in the sequence.

The *DataFormat* object must contain the SVID of the satellite transmitting the navigation data as an *integer*, the WN and TOW at the start of the navigation data page as an *integer*, and the navigation data bits for a nominal page (also called double page) which are 240 bits as a *BitArray* object. The *BitArray* object can be initialized from data in binary format, hex format, integer or any Python Byte Array objects.

Additionally, the *DataFormat* object will take as parameters the signal frequency band and the CRC status. The only signal frequency band supported by the receiver is the E1-B, identified with the string *"GAL_L1BC"*, if no band is specified, the receiver will take that as default. The CRC status is a boolean value that indicated if the page has passed the CRC verification, it is set to *True* by default.

If the custom navigation data is available in Septentrio Binary Format (SBF), the receiver already includes the input iterator *SBFAscii* to handle it. However, the SBF file should be converted to ASCII readable format with the SBF Converter included in the free Septentrio RXTools suite.

If the custom format of the data is not supported by the receiver, a new input iterator should be developed following the instructions in Section IV-A. The new input iterator can be forwarded as a parameter to the receiver as any of the native input iterators without any difference.

B. Receiver Options

The receiver has several configuration parameters that can be defined previous to execution. Those parameters can be specified within the code in a Python dictionary or in a separate JSON File and served as an input parameter to the receiver. The receiver will load default values for the configuration parameters not specified.

The parameters allow to specify the folder and scenario path, and the values of the Merkle Root, Public Key and KROOT. The logs can also be configured, setting a log level for the file logs, the console logs or if it should log at all. Finally, the user can also configure the sync source for the time synchronization check and the tag length required to authenticate data, among other parameters.

V. Data Identification

A. Tag - Data Linkage

On the previous versions of the OSNMA User ICD, the receiver could link the tags received with the navigation data they authenticate using the triplet of parameters ID_{data}:

$$ID_{data} = [PRN_D, ADKD, IOD_{data}] \tag{1}$$

Where PRN_D identifies the satellite which data is being authenticated, $ADKD$ corresponds to the ADKD mask type, and IOD_{data} has different values according to the ADKD value. In the case of ADKD0, the parameter IOD_{data} contains the 3 LSB-truncated I/NAV IOD bits of the authenticated ephemeris and clock data (Words 1, 2, 3 and 4).

This legacy approach had several problems of collision on the IOD truncated value and has been changed on the current version of the OSNMA User ICD [9]. Now, a tag retrieved in a subframe transmitted at the time GST_{SF} is linked with the navigation data transmitted in the previous subframe (at the time $GST_{SF} - 30$ sec). However, there is still room for improvements and optimizations, and that is what *OSNMAlib* does for the ADKD 0 and 12.

B. Implemented Optimization

To optimize the navigation data linkage, *OSNMAlib* tracks the complete 10 bits IOD value of the I/NAV Ephemeris and Clock data words that are authenticated by ADKD 0 and 12, recording the Galileo System Time (GST) value of the first and the last word received with such IOD.

When receiving a tag with ADKD0, the software searches on the list of ADKD0 navigation data blocks for the first block with a start GST lower than the reception subframe GST for the tag, and links them. Using this technique, the tags get linked with the authenticated data using the full value of the IOD and the repetition problem gets solved.

In case the IOD value changes at the same subframe the tag is received, the tag will be linked to the previous subframe data, because the start GST of the new ADKD0 navigation data block identified by the new IOD will not be less than the subframe GST of the tag. In case the IOD remains constant, the tag gets linked to the previous (that is the same as the current) ADKD0 navigation data block.

Moreover, this technique has another advantage. In case the receiver loses some ADKD0 words at the beginning of the subframe previous to receiving the tag, it can effectively use the words from the same subframe where the tag is transmitted to generate the ADKD0 data block (if the IOD is the same) as illustrated in Fig. 6. By using the full IOD field value, the receiver will have no doubts that the regenerated ADKD0 data block contains correct navigation data.

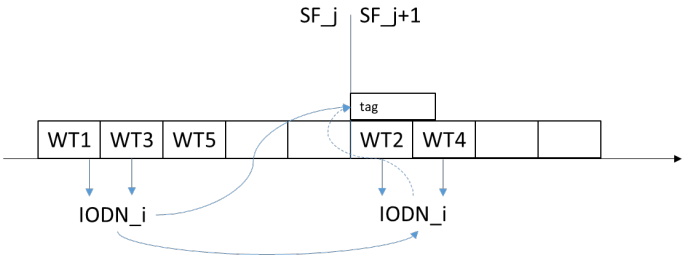

Fig. 6. Using the complete IOD we can use I/NAV words from the same subframe as the tag to perform the authentication.

Using this approach, we can even authenticate navigation data of a satellite that is no longer in view but for which we have stored data by using the cross-authentication tags from other satellites. When a cross-authentication tag for the navigation data of a satellite that is no longer in view fails, then the stored data can not be used anymore for authentication.

The logic of the tag-data linkage algorithm is described in the Fig. 6. The same algorithm is applied for ADKD12.

VI. TESLA Keychain Management

The cornerstone of the OSNMA protocol is the transmission of TESLA keys and authentication of them against the previously transmitted keys. In TESLA, the keys are generated in a chain using a secured one-way hash function and, then, transmitted in reverse order. With this schema, the receiver can authenticate any TESLA key from the chain using an older

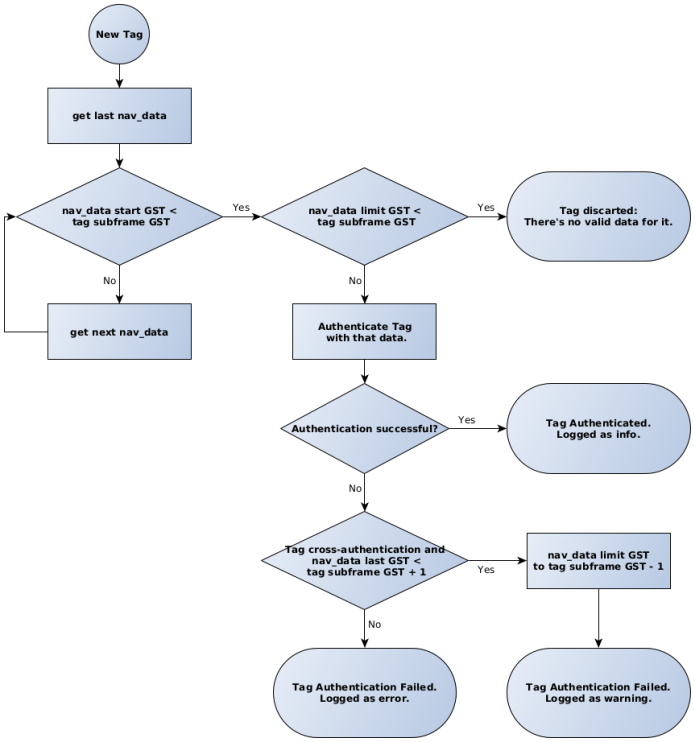

Fig. 7. OSNMA receiver data-tag linkage flowchart.

authenticated key (e.g. the root key transmitted in the DSM-KROOT message) simply by doing a predefined number of hashes.

A. Authenticating TESLA Keys

Therefore, one of the first functionalities that needs to solve the TESLA Chain module is the storage and verification of TESLA keys. The most efficient approach to reduce the amount of operations necessary to verify a TESLA key is to calculate the hashes up to the most recent verified key, instead of calculating all the hashes until the root key.

The most important parameter to know the amount of hashes to be calculated is the index of a received TESLA key. The index of a TESLA key is in reference to the root key, i.e. the number of TESLA keys transmitted since the root key. According to the configuration in the User ICD version 1.0, all the satellites transmit the same TESLA key at the same epoch. Then, the index of a received TESLA key can be computed as:

$$index = \left\lfloor \frac{GST_i - GST_0}{30} \right\rfloor + 1 \qquad (2)$$

Where GST_i is the Galileo System Time at the start of the 30 seconds subframe where the key is received, GST_0 is the Galileo System Time associated with the root key.

B. Storing TESLA Keys

In order to store the TESLA keys, there are two possible approaches: storing all the keys of the chain and serving each

key when needed or saving the last authenticated TESLA key (and the root key for reference) and computing the necessary key when needed.

The first approach, storing all the keys, has an obviously high memory complexity that is more relevant the longer the TESLA chain, stressing the memory requirements for the receivers. But it has a very low computational complexity when serving a key at a given index.

The second approach, storing only the root key and the last authenticated TESLA key, has a very low memory requirement: storing only 2 keys. But it requires to compute all the hashes up to the index of a key when this key is requested.

In a nominal operation of the OSNMA protocol where at least one satellite is always in view, only the last verified key will be requested to authenticate the tags; therefore requiring no extra hash computation. That is because once the receiver gets a key, process all the tags that have been waiting for that key. If the receiver do not miss any key (one satellite in view every 30 seconds subframe), no tag will be requesting and old key.

But even in case the receiver loses sight of all satellites for 30 minutes (and assuming the data authenticated by the tags is still relevant), only 60 hashes have to be done to generate the TESLA Key to authenticate the tags received before the blackout. Taking into consideration that the hash operation is a very efficient operation, this is an assumable price to pay in front of the memory requirements of keeping the full TESLA chain in memory.

The overall process discussed is illustrated in the Figure 8.

C. Handling of Floating Root Keys

Another point of interest related with the TESLA chain is the transmission of floating root keys in the DSM-KROOT message. Floating root keys are keys from the same TESLA chain, but are transmitted in the DSM-KROOT with their signature and the configuration parameters of the chain. Consequently, the change of the root key received does not imply a TESLA chain change, as indicated by the *KROOT Chain ID* parameter, which remains the same.

The idea behind the floating root keys is to facilitate the TESLA key verification process for those receivers starting in the middle of a TESLA Chain, allowing them to perform less hashes. In our implementation of the OSNMA protocol, when a floating DSM-KROOT message is received, it is stored inside the *TESLAChain* object, replacing the last DSM-KROOT stored.

Despite updating the DSM-KROOT value, the root key of the TESLAChain and the parameters regarding the root key are not updated, and keep fixed to the ones in the first DSM-KROOT message received. This is necessary to maintain index consistency on the already assigned key indexes on tags and last received TESLA key. At the end, there is not a loss of efficiency because the last authenticated TESLA key is always more recent than a possible floating root key.

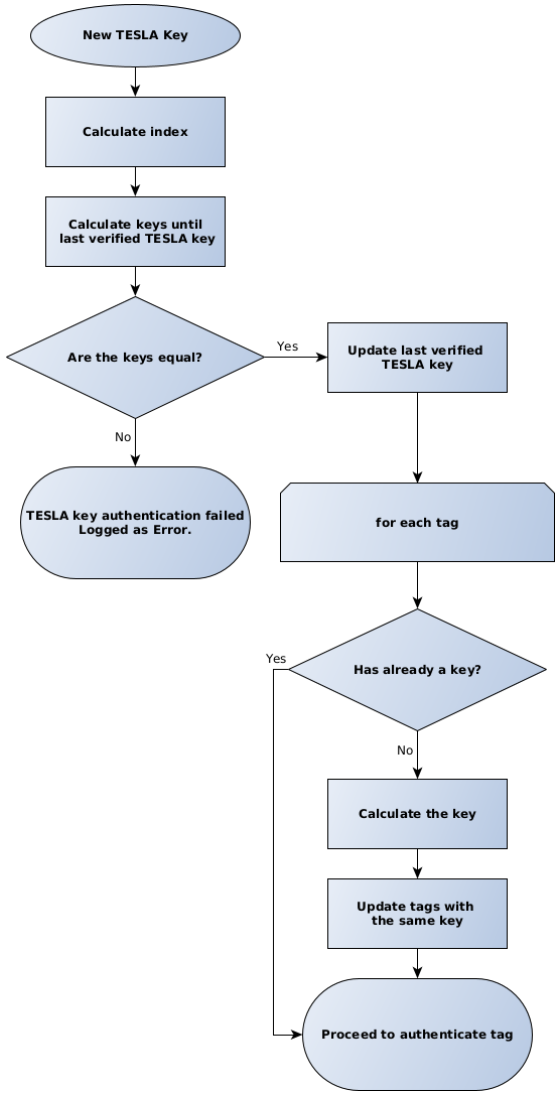

Fig. 8. OSNMA TESLA key management flowchart.

However, the new floating DSM-KROOT is saved to disc as the root key for this chain and replacing the last one, easing the process of a future *Hot Start*.

VII. Subframe Regeneration

One problem with the OSNMA protocol is that its basic structure of information is different from the basic structure of information of the Galileo navigation message. In Galileo I/NAV, each nominal page (2 seconds) is protected with a convolutional Forward Error Correction (FEC) code protecting it from fading and allowing the regeneration of the page even if some bits are lost. In OSNMA the basic structure of information is a subframe (15 nominal pages, 30 seconds), an is not protected with any redundant technique.

While this statement is mostly true regarding the MACK message of OSNMA, for the HKROOT message a more intelligent fine-grained approach can be developed. At a given epoch, all the satellites transmit DSM blocks in the HKROOT

message related to the same whole DSM message. Also, a DSM message is fully transmitted by each satellite before transmitting the next. Finally, the DSM block offset at which each satellite starts transmitting the DSM block sequence is defined by (3).

$$\Delta_{DSM} = mod(PRN_A, N) \tag{3}$$

Where PRN_A identifies the Galileo satellite transmitting the authentication data, and N is currently undefined in [9], although in line with previous test ICD versions and the current test signal, *OSNMAlib* implements $N = 8$. For example, a satellite with PRN_A 3 will start transmitting the DSM message at Block ID (BID) 3, while a satellite with PRN_A 25 will start transmitting the DSM message at BID 1.

The DSM offset was enforced by the Equation 3 in the Internal Test Phase ICD and during the development and test of the *OSNMAlib*. However, the most recent User ICD [9] no longer specifies how the DSM offset value is calculated.

With this information, and knowing that it may vary in the future, it is possible to reconstruct a DSM block from various DSM blocks with the same BID that are missing different pages. This reconstruction can be done even if the page number 2 (the one containing the DSM ID and DSM BID fields) is missing in both of them; as long as it has been retrieved from another DSM block for both subframes.

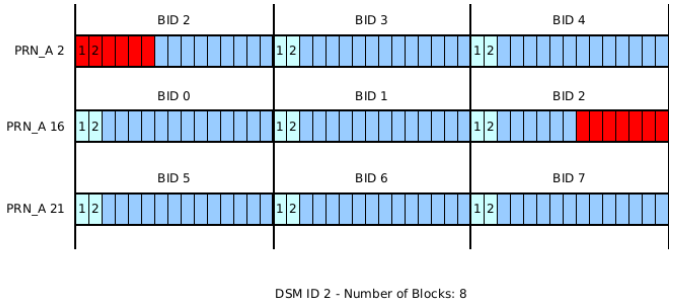

Fig. 9. Regenerate DSM blocks merging pages of half read subframes. In this case, the DSM block 2 can be regenerated.

In order to merge DSM blocks of future subframes is necessary to read the DSM ID of that subframe because it may have changed (for example in the transmission of a floating root key). While often it will be the case, the OSNMA protocol does not commit to align the start of the transmission of a new DSM message with the end of a full transmission of the current DSM.

However, knowing the DSM ID of the current subframe allows to merge current blocks with blocks received in the past. This can be done as long as the number of subframes between the block in the past and the current block is not bigger than the difference between the BID offset for the satellite and the current BID. This way, the receiver can be sure that they belong to the same DSM ID, because the starting BID is fixed for every satellite.

This case may be relevant if we consider the Time To First Authenticated Fix (TTFAF) situation in a warm start. When the receiver starts to receive data, it may start in the middle of a subframe transmission. With the classic approach, that subframe would be discarded because of missed pages. With our approach, if the full DSM message transmission does not start in the first full subframe received (the BID is not the same as the block offset for that satellite), the partially received subframe can be used and merged with the next reading of that subframe.

For example, with only one satellite in view transmitting OSNMA data and the receiver starting after the first page of a subframe, our approach can lead to a 28-second improvement in the TTFAF. With more satellites in view, the effectiveness will depend on the relative BID offset between them. The approach is also extremely relevant in situation with heavy interference or discontinuities in the received signal, where 30 seconds subframes may be discarded for 2 seconds interferences.

Figure 9 illustrates a possible scenario where the receiver starts with 2 OSNMA transmitting satellites in sight and satellite PRN_A 2 enters at the middle of a subframe. Knowing the BID of the other DSM blocks, the receiver deduces that the BID of the half read DSM block is 2. After around 60 seconds, the satellite PRN_A 16 leaves sight in the middle of a subframe. The receiver, knowing that the satellite 16 was transmitting the DSM BID 2, merges the data of both subframes and regenerates the DSM block 2.

VIII. Test Results

As an example of how *OSNMAlib* works we have included the contents of the logs from the execution some tests. As previously indicated, some data was gathered during the first OSNMA testing, before the *Public Observation Phase*. The log snips are selected to show relevant functionality, and are properly dated and explained

A. Configuration 1 Logs

The data used on this test was transmitted on the 04/12/2020 from 11:30 to 12:28; with the TESLA chain Configuration 1 of the OSNMA Receiver Guidelines for the Test Phase, Issue 1.0 [11].

Figure 10 shows how the receiver is decoding subframes from different satellites (i.e. 3, 5, and 9). Some satellites are not transmitting OSNMA data, such as satellite 3. For the satellites transmitting OSNMA data, the receiver logs the DSM ID of the DSM received and its block ID.

After receiving the 8 necessary DSM blocks for the DSM number 7, the DSM-KROOT message is authenticated at WN 1110 and TOW 473550. The Chain ID (CID) of the root key (KROOT) is the same as the CID in force and the Chain and Public Key Status (CPKS) is NOMINAL. Therefore, a new TESLA Chain is created from the DSM-KROOT message and the receiver is set to STARTED state.

Now, the receiver starts decoding and authenticating the MACK messages received while waiting for reconstructing the first DSM-KROOT. In this configuration, there are no FLX

```
INFO - --- SUBFRAME --- WN 1110 TOW 473520 SVID 3 ---
INFO - No OSNMA data.
INFO - --- SUBFRAME --- WN 1110 TOW 473550 SVID 5 ---
INFO - Received block 5 from DSM ID 7.
INFO -  DSM ID 7 blocks: dict_keys([2, 6, 5, 3, 7, 4, 0])

INFO - --- SUBFRAME --- WN 1110 TOW 473550 SVID 9 ---
INFO - Received block 1 from DSM ID 7.
INFO -  DSM ID 7 blocks: dict_keys([2, 6, 5, 3, 7, 4, 0, 1])

INFO - KROOT with CID: 1 - PKID: 1 - GST0: WN 1110 TOW 468000
    AUTHENTICATED

INFO - NMAS is TEST.
INFO - Chain in force is 1.
INFO - CPKS is NOMINAL.

INFO - Start status from WARM_START to STARTED.
INFO - MACSEQ AUTHENTICATED
    PRN_A: 5 GST_SF: 1110 473460 TAGS ADDED: 0
INFO - MACSEQ AUTHENTICATED
    PRN_A: 9 GST_SF: 1110 473460 TAGS ADDED: 0
INFO - MACSEQ AUTHENTICATED
    PRN_A: 24 GST_SF: 1110 473460 TAGS ADDED: 0
```

Fig. 10. Authentication of a KROOT.

tags, therefore the authentication of the MACSEQ message adds no new tags to the tag pool.

Figure 11 shows the subframe transmitted by satellite 5 at WN 1110 and TOW 473580. The TESLA key received in that subframe allows to authenticate the MACSEQ from the previous subframe, and all the tags that were waiting for this key.

At this moment, only three satellites in view are transmitting OSNMA data, therefore there are only three MACSEQ authentications and three TAG0 authentications. However, the Tags are also cross-authenticating the other satellites that are in view but are no transmitting OSNMA data (i.e. satellite 31 and 3).

Figure 12 shows how *OSNMAlib* logs the navigation data that can be considered authenticated after receiving enough number of tags. For ADKD 0 (and 12) it logs the satellite from which navigation data is being authenticated, the IOD of the data to ease identification and the range of GST for which the data has been authenticated. Because for ADKD 4 there's no IOD in the words it authenticates nor is specific to a particular satellite, we just log the range of GST for which data of those words have been authenticated.

After processing the first subframe received at GST 1110-473580 and authenticating the possible tags and data, the subframes received on the same GST from other satellites (i.e. satellite 9 and 24) do not allow us to authenticate any more tags or data. That is because all satellites are transmitting the same key at the same epoch.

```
INFO - --- SUBFRAME --- WN 1110 TOW 473580 SVID 5 ---
INFO - Received block 6 from DSM ID 7.
INFO -  DSM ID 7 blocks: dict_keys([0, 6])

INFO - MACSEQ AUTHENTICATED
    PRN_A: 5 GST_SF: 1110 473550 TAGS ADDED: 0
INFO - MACSEQ AUTHENTICATED
    PRN_A: 9 GST_SF: 1110 473550 TAGS ADDED: 0
INFO - MACSEQ AUTHENTICATED
    PRN_A: 24 GST_SF: 1110 473550 TAGS ADDED: 0
INFO - Tag verification:

INFO - Tag AUTHENTICATED
    (5, 0, 4) PRN_A: 5 GST_SF: 1110 473550 TAG0
INFO - Tag AUTHENTICATED
    (9, 0, 4) PRN_A: 5 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (31, 0, 3) PRN_A: 5 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (255, 4, 0) PRN_A: 5 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (3, 0, 3) PRN_A: 5 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (9, 0, 4) PRN_A: 9 GST_SF: 1110 473550 TAG0
INFO - Tag AUTHENTICATED
    (31, 0, 3) PRN_A: 9 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (24, 0, 4) PRN_A: 9 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (255, 4, 0) PRN_A: 9 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (5, 0, 4) PRN_A: 9 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (24, 0, 4) PRN_A: 24 GST_SF: 1110 473550 TAG0
INFO - Tag AUTHENTICATED
    (9, 0, 4) PRN_A: 24 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (31, 0, 3) PRN_A: 24 GST_SF: 1110 473550
INFO - Tag AUTHENTICATED
    (255, 4, 0) PRN_A: 24 GST_SF: 1110 473550
INFO - Data authenticated:

INFO - AUTHENTICATED: ADKD 0 - Satellite 5 - IOD 0b0000010100
```

Fig. 11. Authentication of the MACSEQ and Tags.

B. End of Chain Logs

The data used on this test was transmitted on the 18/03/2021 from 12:07 to 13:06; with the TESLA chain Configuration 6 before transitioning to Configuration 1.

Figure 13 shows that a DSM-KROOT is authenticated at WN 1125 and TOW 385830; and it is identified with CID 3. However, the CID in force is 2 and the CPSK is EOC; that means that the receiver has read and authenticated a DSM-KROOT message belonging to the next TESLA chain to be in force. That happens because during the EOC phase both the DSM-KROOT for the current chain and the DSM-KROOT for the next chain are transmitted interleaved.

```
INFO - Data authenticated:

INFO - AUTHENTICATED: ADKD 0 - Satellite 5 - IOD 0b0000010100
         GST 1110-473490  to  GST 1110-473550
         Words 1, 2, 3, 4, 5

INFO - AUTHENTICATED: ADKD 0 - Satellite 9 - IOD 0b0000010100
         GST 1110-473490  to  GST 1110-473550
         Words 1, 2, 3, 4, 5

INFO - AUTHENTICATED: ADKD 0 - Satellite 31 - IOD 0b0000010011
         GST 1110-473490  to  GST 1110-473550
         Words 1, 2, 3, 4, 5

INFO - AUTHENTICATED: ADKD 0 - Satellite 3 - IOD 0b0000010011
         GST 1110-473490  to  GST 1110-473550
         Words 1, 2, 3, 4, 5

INFO - AUTHENTICATED: ADKD 0 - Satellite 24 - IOD 0b0000010100
         GST 1110-473490  to  GST 1110-473550
         Words 1, 2, 3, 4, 5

INFO - AUTHENTICATED: ADKD 4 - Satellite Any
         GST 1110-473550  to  GST 1110-473550
         Words 6, 10

INFO - --- SUBFRAME --- WN 1110 TOW 473580 SVID 9 ---
INFO - Received block 2 from DSM ID 7.
INFO -  DSM ID 7 blocks: dict_keys([0, 6, 2])

INFO - Tag verification:

INFO - Data authenticated:

INFO - --- SUBFRAME --- WN 1110 TOW 473580 SVID 24 ---
INFO - Received block 1 from DSM ID 7.
INFO -  DSM ID 7 blocks: dict_keys([0, 6, 2, 1])

INFO - Tag verification:

INFO - Data authenticated:
```

Fig. 12. Navigation data with enough bits to be considered authenticated.

In this case, *OSNMAlib* stores the read and authenticated DSM-KROOT and keeps processing data hoping to find the current DSM-KROOT.

Figure 14 shows how after some subframes, a new DMS-KROOT is retrieved and authenticated. In this case is the one belonging to current TESLA chain in force, so it is used and the receiver is set to STARTED.

C. New Public Key Logs

The data used on this test for the first step of the NPK process was transmitted on the 25/01/2021 from 12:04 to 13:03; with the TESLA chain Configuration 2 and authenticated with the Public Key ID 3.

The data used on this test for the second step of the NPK process was transmitted on the 26/01/2021 from 10:23 to 11:22; with the TESLA chain Configuration 2 and authenticated with the Public Key ID 4.

```
INFO - --- SUBFRAME --- WN 1125 TOW 385830 SVID 3 ---
INFO - Received block 7 from DSM ID 3.
INFO -  DSM ID 3 blocks: dict_keys([1, 3, 0, 2, 4, 5, 6, 7])

INFO - KROOT with CID: 3 - PKID: 9 - GST0: WN 1125 TOW 370800
    AUTHENTICATED

INFO - NMAS is TEST.
INFO - Chain in force is 2.
INFO - CPKS is EOC.

INFO - --- SUBFRAME --- WN 1125 TOW 385830 SVID 8 ---
INFO - No OSNMA data.
```

Fig. 13. EOC event with a DSM-KROOT retrieved that is not in force.

```
INFO - --- SUBFRAME --- WN 1125 TOW 386040 SVID 3 ---
INFO - Received block 6 from DSM ID 2.
INFO -  DSM ID 2 blocks: dict_keys([0, 2, 7, 1, 3, 4, 5, 6])

INFO - KROOT with CID: 2 - PKID: 9 - GST0: WN 1125 TOW 370800
    AUTHENTICATED

INFO - NMAS is TEST.
INFO - Chain in force is 2.
INFO - CPKS is EOC.

INFO - Start status from WARM_START to STARTED.
INFO - MACSEQ AUTHENTICATED
    PRN_A: 25 GST_SF: 1125 385680 TAGS ADDED: 0
```

Fig. 14. EOC event with a DSM-KROOT retrieved that is in force.

Figure 15 shows how a DSM-KROOT in force is authenticated with the Public Key ID 3 while the CPKS value is set to NPK.

```
INFO - --- SUBFRAME --- WN 1118 TOW 129690 SVID 19 ---
INFO - Received block 0 from DSM ID 0.
INFO -  DSM ID 0 blocks: dict_keys([5, 6, 1, 4, 2, 7, 3, 0])

INFO - DSM ID 0 number of blocks: 8

INFO - KROOT with CID: 2 - PKID: 3 - GST0: WN 1118 TOW 115200
    AUTHENTICATED

INFO - NMAS is TEST.
INFO - Chain in force is 2.
INFO - CPKS is NPK.

INFO - MACSEQ AUTHENTICATED
    PRN_A: 19 GST_SF: 1118 129660 TAGS ADDED: 0
INFO - MACSEQ AUTHENTICATED
```

Fig. 15. DSM-KROOT authenticated with a PKID of 3. The CPKS value is NPK

Figure 16 shows that a DSM-PKR message is transmitted along with the DSM-KROOT message, as expected on the

first step of the NPK process. This DSM-PKR is retrieved and authenticated. The PKID of the new Public Key is 4 but the Public Key now in use is the 3. Therefore, the Public Key is stored in the receiver waiting for the key change.

```
INFO - --- SUBFRAME --- WN 1118 TOW 128730 SVID 30 ---
INFO - Received block 0 from DSM ID 13.
INFO -  DSM ID 13 blocks: dict_keys([3, 2, 6, 1, 4, 7, 5, 8, 9, 10, 11, 12, 0])

INFO - DSM ID 13 number of blocks: 13

INFO - PKR with NPKID 4 verified.
INFO - New PK. Current PKs: dict_keys([3, 4])
INFO - --- SUBFRAME --- WN 1118 TOW 128730 SVID 7 ---
```

Fig. 16. New Public Key read with PKID of 4.

Figure 17 shows the moment where the new Public Key ID 4 enters in force. That occurs by receiving a DSM-KROOT message that uses the new Public Key for it is authentication. If the DSM-KROOT is successfully authenticated, the Public Key in force is changed to the new key and the last one is deleted.

It is also worth it to notice that a change of Public Key does not imply a change of Tesla Chain. If we compare Figures 15 and 17, we can check that the TESLA Chain in force keeps at the number 2.

```
INFO - --- SUBFRAME --- WN 1118 TOW 206700 SVID 5 ---
INFO - Received block 4 from DSM ID 1.
INFO -  DSM ID 1 blocks: dict_keys([5, 0, 7, 2, 6, 1, 3, 4])

INFO - KROOT with CID: 2 - PKID: 4 - GST0: WN 1118 TOW 205200
    AUTHENTICATED

INFO - NMAS is TEST.
INFO - Chain in force is 2.
INFO - CPKS is NPK.

INFO - PK in force changed from 3 to 4
INFO - Tag verification:

INFO - Data authenticated:

INFO - --- SUBFRAME --- WN 1118 TOW 206700 SVID 24 ---
INFO - Received block 7 from DSM ID 1.
```

Fig. 17. DSM-KROOT authenticated with a PKID of 4. The CPKS value is NPK

D. Public Key Revocation Logs

The data used on this test for the first step of the Public Key Revocation (PKREV) process was transmitted on the 29/01/2021 from 09:24 to 10:23; with the TESLA chain Configuration 2 and authenticated with the Public Key ID 4.

The data used on this test for the second step of the PKREV process was transmitted on the 29/01/2021 from 11:32 to 12:31; with the TESLA chain Configuration 2 and authenticated with the Public Key ID 5.

Figure 18 shows the authentication of a DSM-KROOT with CID 2 and PKID 5. The NMAS is set to DONT_USE and the CPKS is PKREV. Therefore, the previous PKID 4 is revoked. While the NMAS is set to DONT_USE the tags are not processed.

```
INFO - --- SUBFRAME --- WN 1118 TOW 464910 SVID 2 ---
INFO - Received block 5 from DSM ID 2.
INFO -  DSM ID 2 blocks: dict_keys([0, 2, 6, 7, 1, 3, 4, 5])

INFO - KROOT with CID: 2 - PKID: 5 - GST0: WN 1118 TOW 460800
    AUTHENTICATED

INFO - NMAS is DONT_USE.
INFO - Chain in force is 1.
INFO - CPKS is PKREV.

INFO - Public Key 4 revoked.
INFO - Fallback to COLD_START from STARTED
WARNING - NMA Status: Don't Use. Subframe tags not processed.
INFO - --- SUBFRAME --- WN 1118 TOW 464910 SVID 3 ---
INFO - No OSNMA data.
```

Fig. 18. DSM-KROOT authenticated with a PKID of 5. The CPKS value is set to NPK and the NMAS to DONT_USE

Figure 19 shows how, during the DONT_USE status, a DSM-PKR with the new key is regularly transmitted to authenticate the DSM-KROOT. This is necessary because the only way for the receiver to know the current state of the OSNMA protocol is to authenticate a DSM-KROOT message.

```
INFO - --- SUBFRAME --- WN 1118 TOW 465180 SVID 7 ---
INFO - Received block 11 from DSM ID 13.
INFO -  DSM ID 13 blocks: dict_keys([0, 12, 1, 5, 6, 2, 7, 3, 8, 4, 9, 10, 11])

INFO - PKR with NPKID 5 verified.
INFO - New PK. Current PKs: dict_keys([5])
INFO - Start status from COLD_START to WARM_START
WARNING - NMA Status: Don't Use. Subframe tags not processed.
INFO - --- SUBFRAME --- WN 1118 TOW 465180 SVID 3 ---
INFO - No OSNMA data.
```

Fig. 19. While the DONT_USE status, a DSM-PKR is tranmitted with PKID of 5

Figure 20 shows how eventually the NMAS is set again to TEST, allowing navigation authentication from this moment onward. Note that the CPKS is still PKREV, because the new Public Key will still be transmitted during a defined period.

IX. Summary and Conclusions

This paper has presented *OSNMAlib*, an open source Python library implementing Galileo OSNMA, for its integration in receivers and applications.

OSNMAlib architecture contains the necessary classes to process OSNMA data and perform the Galileo I/NAV data authentication. The software workflow is organised around an *OSNMAReceiver* class that handles all the OSNMA protocol actions. It is supported by a GNSS receiver and data decoder class, that takes the I/NAV pages in decoded SBF, *hex* or other

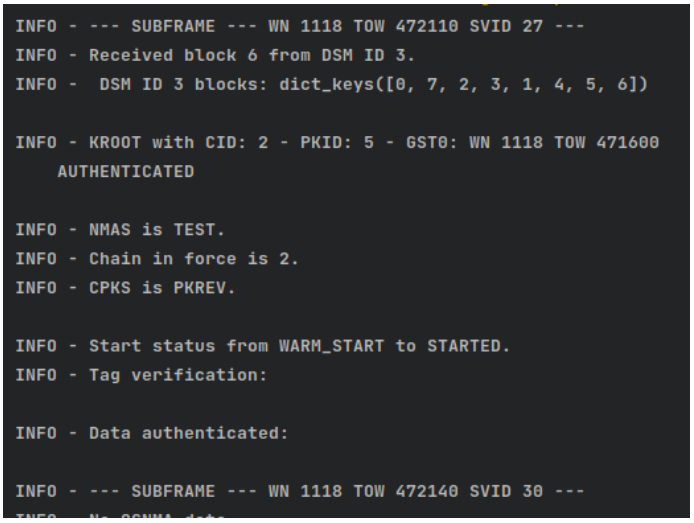

Fig. 20. When the NMAS returns to TEST, the DMS-KROOT messages are authenticated with PKID 5

formats, and other classes to manage the OSNMA overall status, keychain, public keys and Merkle tree, data-tag allocation and data authentication status. While the position calculation is not performed, it reports to the receiver which data and from which satellites is authenticated, allowing to introduce OSNMA in the receiver positioning workflow. *OSNMAlib* can also handle the public key and chain renewal and revocation processes.

At startup, *OSNMAlib* reads the available configuration and first checks the loose time synchronization assumptions and time source, when required. It then verifies the availability of the Merkle root, public key, and root key, and starts the logic to obtain them if they are not available, as per the currently applicable OSNMA SIS ICD. *OSNMAlib* has been also used for testing some optimizations of the OSNMA receiver logic, such as linking I/NAV pages from different subframes and authenticating them as early as possible with the keys and tags received, reducing the time to authenticated fix.

The *OSNMAlib* package is provided with test scenarios based on data collection during the OSNMA internal and public observation test phases. A procedure for running the tests is presented as an example. The available tests include various configurations and chain renewal, public key renewal, and public key revocation procedures.

Further development and testing of *OSNMAlib* will include test campaigns with more data, optimization of the code execution, additional modules integrating other receiver blocks, and real-time performance evaluation. While the platform has mostly been tested in post-processing, it should allow real-time OSNMA processing in operational receivers.

During the so-called OSNMA *Public Observation Phase*, currently ongoing, the OSNMA signal is still transmitted in *test* mode, but with high continuity and availability. We hope that *OSNMAlib* can be a useful tool for understanding OSNMA, reducing the effort of integrating it in receivers and applications, and eventually increasing PNT resilience at no or minimal nuisance for users and applications.

Acknowledgment

This work has been supported by European Commission contract SI2.823546/9 and by the Spanish Ministry of Science and Innovation project PID2020-118984GB-I00.

References

[1] "Tests of Galileo OSNMA underway," https://ec.europa.eu/defence-industry-space/tests-galileo-osnma-underway-2021-02-11_en, accessed: 2021-11-06.

[2] T. E. Humphreys, "Detection strategy for cryptographic GNSS anti-spoofing," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 2, pp. 1073–1090, 2013.

[3] G. Seco-Granados, D. Gomez-Casco, J. A. Lopez-Salcedo, and I. Fernandez-Hernandez, "Detection of replay attacks to GNSS based on partial correlations and authentication data unpredictability," *Gps Solutions*, vol. 25, no. 2, pp. 1–15, 2021.

[4] I. Fernandez-Hernandez, V. Rijmen, G. Seco-Granados, J. Simon, I. Rodriguez, and J. D. Calle, "A navigation message authentication proposal for the Galileo open service," *NAVIGATION, Journal of the Institute of Navigation*, vol. 63, no. 1, pp. 85–102, 2016.

[5] A. Galan, I. Fernandez-Hernandez, G. Seco-Granados, "OSNMAlib," https://github.com/Algafix/OSNMA, 2021.

[6] B. Motella, M. T. Gamba, and M. Nicola, "A real-time OSNMA-ready software receiver," in *Proceedings of the 2020 International Technical Meeting of The Institute of Navigation*, 2020, pp. 979–991.

[7] C. Sarto, O. Pozzobon, S. Fantinato, S. Montagner, I. Fernandez-Hernandez, J. Simon, J. D. Calle, S. C. Diaz, P. Walker, D. Burkey *et al.*, "Implementation and testing of OSNMA for Galileo," in *Proceedings of the 30th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2017)*, 2017, pp. 1508–1519.

[8] "OSNMA: the latest in GNSS anti-spoofing security," https://www.septentrio.com/en/learn-more/insights/osnma-latest-gnss-anti-spoofing-security, accessed: 2021-11-06.

[9] "OSNMA User ICD for the Test Phase, Issue 1.0," European Union, Tech. Rep., Nov 2021.

[10] H. Krekel. (2015, Sep.) Pytest: Python test framework. [Online]. Available: https://docs.pytest.org

[11] "OSNMA Receiver Guidelines for the Test Phase, Issue 1.0," European Union, Tech. Rep., Nov 2021.