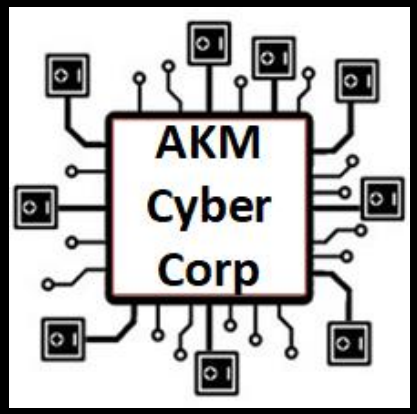


Providing Integrity & Authentication without Exposing Secrets

What is Asset Integrity Management (AIM)?

Asset Integrity Management (AIM) – Asset Integrity Management (AIM) uses an HMAC-SHA256 based methodology providing for a given system, to securely identify and authenticate:

1) Physical (hardware) assets (embedded devices) and virtual (electronic images) assets residing within each physical asset (i.e., the embedded devices and its firmware, software, and data files).
2) A subsystem, consisting of individual assets (primarily physical assets, but could be groups of virtual assets).
3) A system, consisting of a hierarchical collection of subsystems (e.g., an automotive vehicle, train, plane, manufacturing robot, etc.)

These capabilities are applied to systems, subsystems and individual assets from initial creation, through deployment and until end-of-life.

The end goal is to provide a capability to identify, authenticate, and track every element of a system within a hierarchical framework (i.e., first starting within the device level, then moving to the entire device, then the subsystem level, then, if applicable, for collections of subsystems, and finally, the entire system.

AIM Calculations: Electronic Image and the File Asset Tag

What is an Electronic Image?

For the purpose of this document, an “electronic image” is any set of data whose sequential, aggregate set of data, constitutes a logical grouping of electronic data regardless if that grouping is a single text file, an executable file, a binary data file, a container, or other logical sequential grouping of electronic data.

File Asset Tag – At a minimum, the File Asset Tag will contain:

- AIM Identifier of the File (default size of this is 128-bits, 16-Bytes) – This is an implementation unique identifier. The exact format of this identifier is TBD.
- File Type (ex. binary data file, executable, container, FPGA, etc.)
- File Length – number of bytes contained within the file

Electronic Image Example and FIC Calculation

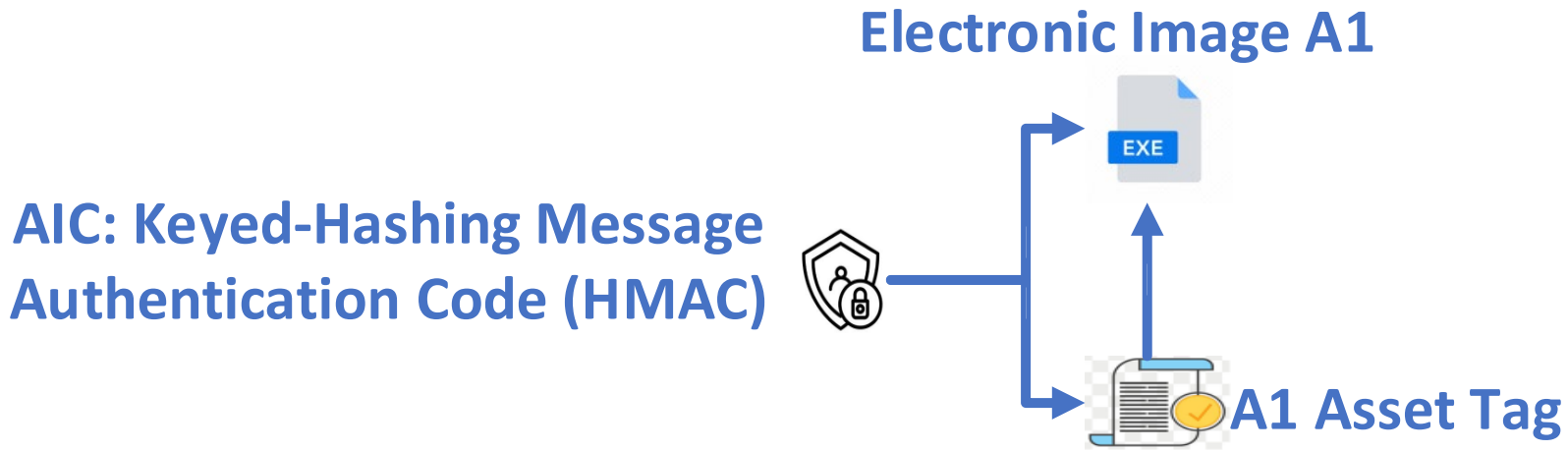
File Integrity Code (**FIC**)– This is the 256-bit, SHA256 (or greater, ex. SHA512), resultant HMAC run over the electronic image component and its associated asset tag and maintained as part of the BackEnd Server's configuration management system. The below mathematical function specifies the calculation for a FIC:

- FIC = HMAC (SHA($AssetTag_{ImgX}$) || SHA(IMG_X), $HMAC\text{-}SECRET_{FIC}$)

Where:

- $AssetTag_{ImgX}$, represents the Asset Tag of Image X.
- IMG_X, represents the Image labeled, component 'X'.
- $HMAC\text{-}SECRET_{FIC}$, represents the HMAC Secret used to Calculate the HMAC.

Values unique to individual files will be used to validate the authenticity of an individual file. If the value of the file does not equal the expected value, then the file can be considered to be compromised (and not the file it claims to be).



AIM Calculations: Device Asset Tag and Device Integrity Code

Endpoint Device Asset Tag – At a minimum, the Endpoint Device Asset Tag will contain the AIM Identifier of the Endpoint Device, the Endpoint Device type, the associated unique hardware identifier of the device (could even be from an onboard Physically Unclonable Device (PUF), and a list of AIM File Identifiers of the file images within the device).

Endpoint Integrity Code (**EPIC**)– This is the 256-bit, SHA256 (or greater, ex. SHA512), resultant HMAC run over an aggregate of information directly associated with a device, including all of the electronic image components that are specified to be part of a device by the BackEnd Server's configuration management system and the Endpoint Device Asset Tag. The below mathematical function specifies the calculation for an EPIC:

- $EPIC = HMAC\ (SHA(AssetTag_{Device}) \mid\mid FIC_{IMG1} \mid\mid FIC_{IMG2} \mid\mid FIC_{IMG3} \ldots, HMAC\text{-}SECRET_{EPIC})$

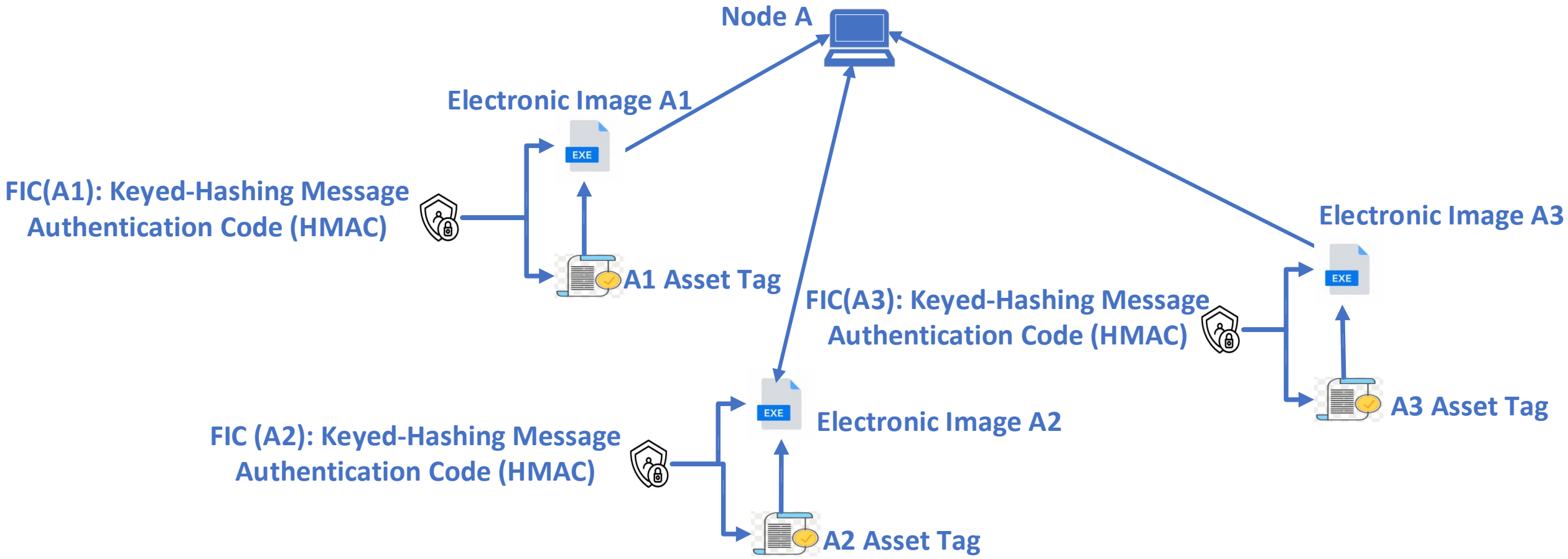
Where:

- $AssetTag_{Device}$, represents the Asset Tag for the local Device.
- IMG_n, represents the Image for the nth component.
- $HMAC\text{-}SECRET_{DIC}$, represents the HMAC Secret used to Calculate the HMAC of the Device.

Values unique to individual devices will be used to validate the authenticity of an individual device. If the value on the device does not equal the expected value, then the hardware can be considered to be compromised (and not the device it claims to be).

Device Example and EPIC Calculation

$$EPIC_{Node\text{-}A} = HMAC\ (SHA(AssetTag_{Node\text{-}A}) \,||\, FIC_{EI\text{-}A1} \,||\, FIC_{EI\text{-}A2} \,||\, FIC_{EI\text{-}A3},\ HMAC\text{-}SECRET_{Node\text{-}A})$$



AIM Calculations: Subsystem Asset Tag and Subsystem Integrity Code

Subsystem Asset Tag – At a minimum, the Subsystem Asset Tag will contain the AIM Identifier of the Subsystem, the Subsystem type (implementation defined), and a list of AIM identifiers of all other nodes within the subsystem or if a Node is representing a subsystem, the AIM identifier of the Subsystem.

Subsystem Integrity Code (**SSIC**)– This is the 256-bit, SHA256 (or greater, ex. SHA512), resultant HMAC run over an aggregate of information associated with all of the nodes within the subsystem that "directly connected" to the Master Subsystem Node and indirectly to all nodes within the subsystem that are "not directly connected to the Subsystem Master Node but which reside within the overall subsystem hierarchy.

The below mathematical function specifies the calculation for a SSIC:

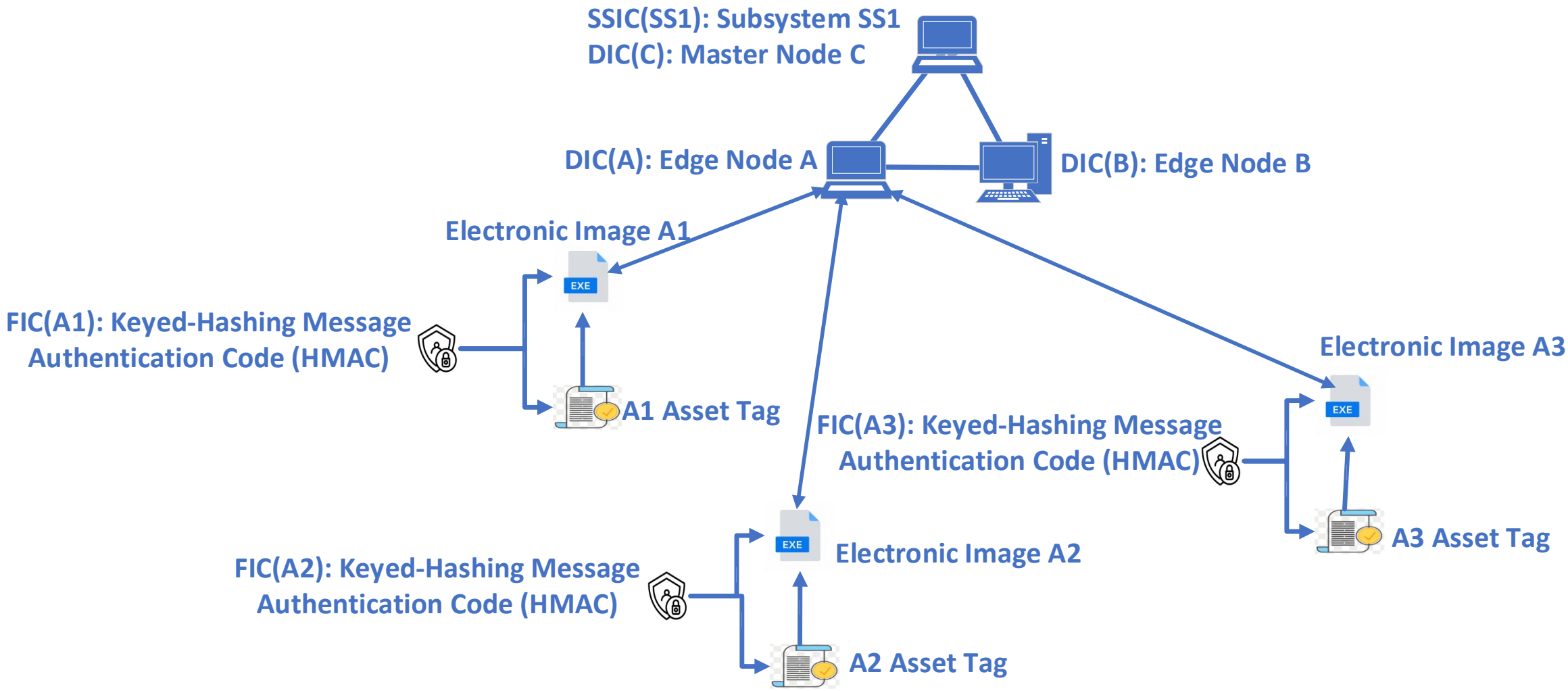
- $SSIC = HMAC\ (SHA(AssetTag_{Subsystem}) \mid\mid IC_1 \mid\mid IC_2 \mid\mid IC_3) \ldots, HMAC\text{-}SECRET_{SSIC})$

Where:

- $AssetTag_{Subsystem}$, represents the Asset Tag for the Subsystem.
- IC_n, represents the Integrity Code used for each of the devices within the relationship. In most cases, the Integrity Code used will be a DIC, but in those instances in which a device is a master device for a lower subsystem, then the IC used will be the SSIC for which the lower subsystem represented by that master device.
- $HMAC\text{-}SECRET_{SSIC}$, represents the HMAC Secret used to calculate the HMAC of the Subsystem.

Subsystem Example and SSIC Calculation

$$SSIC(SS1) = HMAC\ (SHA(AssetTag_{SS1})\ ||\ EPIC_A\ ||\ EPIC_B\ ||\ EPIC_C),\ HMAC\text{-}SECRET_{SS1})$$



AIM Calculations: System Asset Tag and System Integrity Code

System Asset Tag – At a minimum, the System Asset Tag will contain the AIM Identifier of the System, the System type (implementation defined), and a list of AIM identifiers of all other nodes within the system or if a node is representing a subsystem, the AIM identifier of the subsystem it is representing.

System Integrity Code (**SSIC**)– This is the 256-bit, SHA256 (or greater), resultant HMAC run over an aggregate of information associated with all of the nodes within the system that "directly connected" to the Master System Node and indirectly to all nodes within the system that are "not directly connected to the System Master Node but which reside within the overall system hierarchy.

The below mathematical function specifies the calculation for a SSIC:

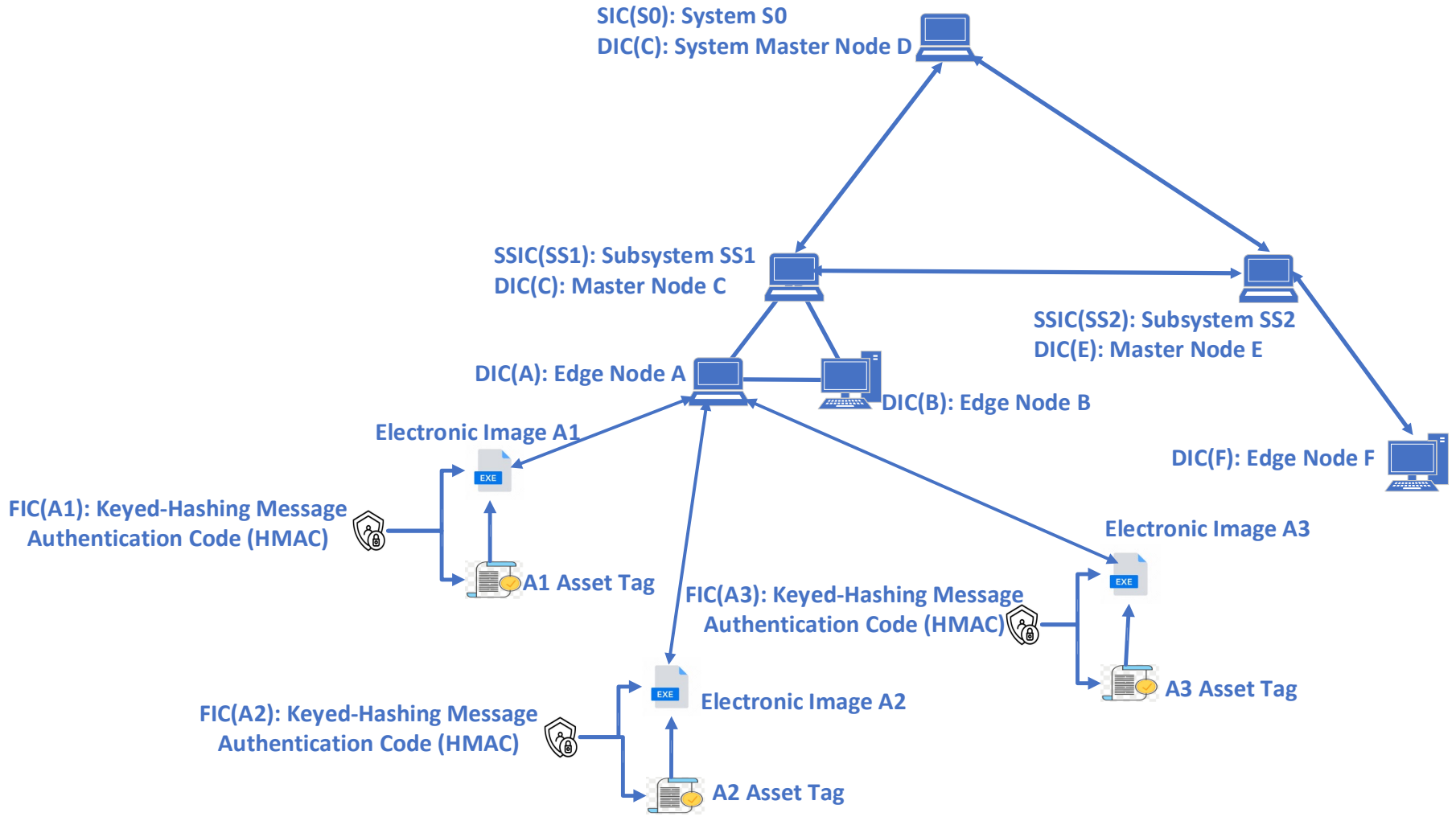
- $SIC = HMAC\ (SHA(AssetTag_{System}) \mid\mid IC_1 \mid\mid IC_2 \mid\mid IC_3) \ldots, HMAC\text{-}SECRET_{SIC})$

Where:

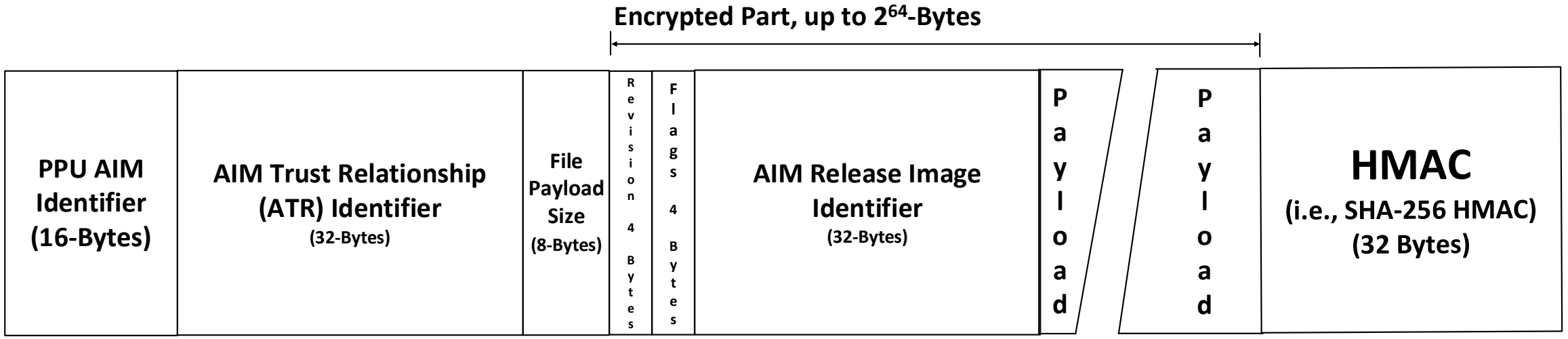
- $AssetTag_{System}$, represents the Asset Tag for the System.
- IC_n, represents the Integrity Code used for each of the devices within the relationship. In most cases, the Integrity Code used will be a DIC, but in those instances in which a device is a master device for a lower subsystem, then the IC used will be the SSIC for which the lower subsystem represented by that master device.
- $HMAC\text{-}SECRET_{SIC}$, represents the HMAC Secret used to calculate the HMAC of the System.

System Example and SIC Calculation

$$SIC(S0) = HMAC\ (SHA(AssetTag_{S0})\ ||\ DIC_D\ ||\ SSIC_{SS1}\ ||\ SSIC_{SS2}),\ HMAC\text{-}SECRET_{S0})$$



Asset Integrity Management (File Structure, via PPU)



Fields:

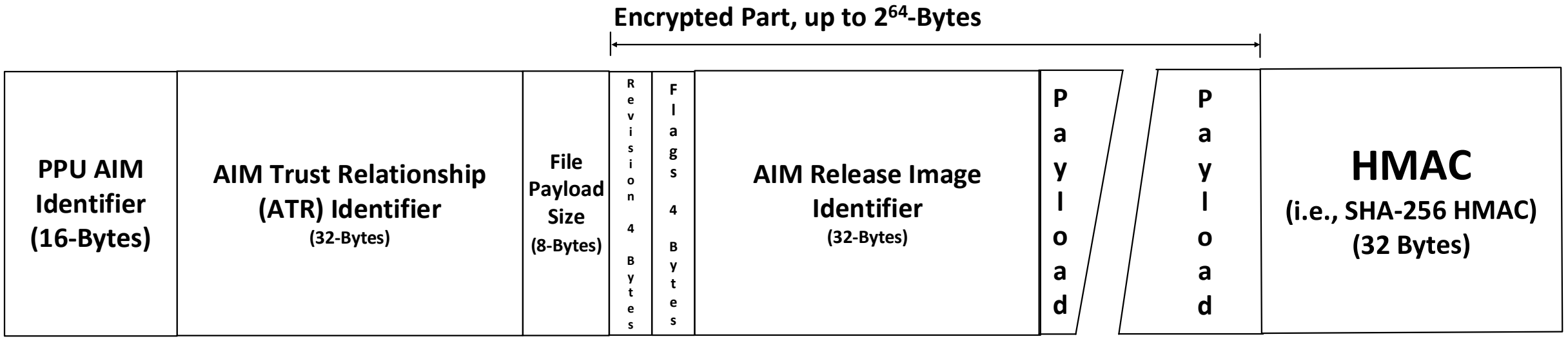
Portable Provisioning Unit (PPU) AIM Identifier – This is the AIM Identifier of the PPU (the management device provisioning the target module).

AIM Trust Relationship Identifier – A 256-bit AIM Identifier that specifically refers to the AIM Trust Relationship (ATR) representing the combination of the firmware release and the specific hardware the release is being bound to.

File Payload Size – This 8-byte field represents the total size of the File Payload, which is limited to 2^{64} minus the size of the fields within the encrypted part of the file structure.

Revision Number – At present, this is 4-bytes, but can easily be modified if necessary and/or requested.

Asset Integrity Management (File Structure, via PPU)



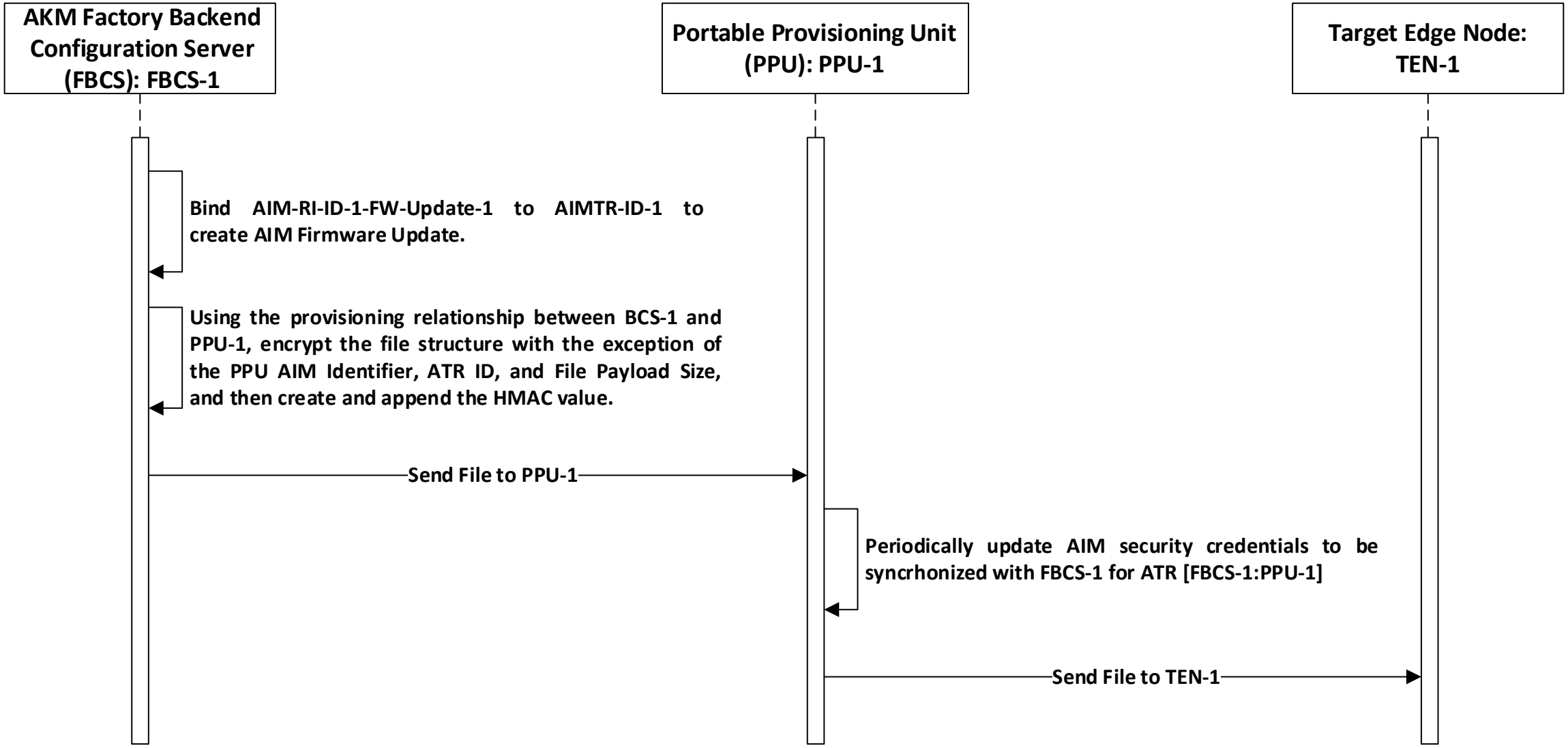
Fields (continued):

AIM Release Image Identifier – A 256-bit AIM Identifier that is specific to the hardware model, but unlike the AIM Security Relationship Identifier, is not specific to the specific hardware. Meaning, all hardware of this type could have the same identical AIM Release Image Identifier.

Payload– Up to 2^{64}-bytes minus the size of the other fields within the encrypted portion of the file structure. This field is certainly made up of multiple other fields, identifiers, and sub-fields, but for the purpose of this high-level overview, subfields and files contained within this field are not relevant.

Asset Tag – A 256-bit Sha-256 HMAC value that ensures the integrity of the entire file structure. Thus, insuring all fields, including the non-encrypted fields have not been tampered with.

Software update from the FBCS (via a PPU)



Software update from the FBCS (via a PPU)

Assumptions:

1) Factory Backend Configuration Server (BCS) has AIM ID: FBCS-1
2) Portable Provisioning Unit (PPU) has AIM ID: PPU-1
3) Target Edge Node has AIM ID: TEN-1
4) AIM Release Image Identifier has AIM ID: AIM-RI-ID-1
5) AIM Trust Relationship Identifier has AIM ID: AIMTR-ID-1
6) AIM Release Image AIM-RI-ID-1's new Revision # is: AIM-RI-ID-1-FW-Update-1
7) The Portable Provisioning Unit MUST store and process the secure information within an HSM that is tamper resistant.
8) The Portable Provisioning Unit is NOT a laptop, but rather an embedded device specifically created for the sole purpose of updating individual edge nodes and entire AIM networks in the field.
9) The PPU can connect to the FBCS prior to receiving the update.

Secure Boot Implementation

The implementation is divided into three parts:

1) HW dependent Secure Boot (optional);
2) Bootloader with main AIM Secure Boot functionality;
3) Hardware Secure Enclave.

The Boot Procedure uses an Encrypted File to store all required system configuration parameters (example: the host device's unique "hardware identifier" (could be via a PUF) and digital signatures of user application images).

Secure Boot Implementation

Hardware Dependent Secure Boot

The HW dependent Secure Boot is an optional part of the Secure Boot Process and must be directly supported by the target microprocessor. Details of the implementation of a secure boot process, varies from manufacturer to manufacturer, but the main concept is virtually always the same – that is, it (the host processor unit) delivers functionality of a hardware Root-of-Trust that can be used to initiate a chain of trusted and encrypted binary images running on the host device. In the solution presented within this document, the HW dependent Secure Boot is responsible for confidentiality and integrity of the stage-1 bootloader binary image.

In the event that the selected microprocessor does not support HW Secure Boot, then this step will be omitted.

Bootloader with AIM Secure Boot Functionality

The standard bootloader that is supported by the target microprocessor has the additional functionality that it is responsible for checking the confidentiality and integrity of selected components within the host device. Thus, simultaneously, providing implicit device authentication of the AIM Identifier and critical system digital components.

Hardware Secure Enclave

This is a specialized hardware device that is responsible for performing cryptographic operations to validate system components or software images. A hardware secure enclave stores all cryptographic credentials within a physically protected external memory.

Application Authentication, Revocation, and Timed Expiration

1) Each Application will have an associated AIM Identifier.
2) Each Application shall have an associated HMAC Secret that is unique to it.
3) The Application's HMAC Secret shall never be exposed external to the Hardware Secure Enclave (HSE).
4) The Application, the Application's Asset Tag[1] (which includes the AIM Identifier), the Application's image address in memory, and the length in bytes, shall all be submitted to the HSE for authentication via an API provided in the interface SDK for the HSE.
5) A Boolean value shall be returned by the HSE via the aforementioned API in (4) above, indicating success or failure of the authentication process. A value of zero or null, will indicate success, while a positive value shall serve as a "token" for the Application. This "token" shall be valid for a preconfigured amount of time in accordance to policy. Once that Application's token has expired, the Application will require resubmission of its information to once again be authenticated. If authentication expiration occurs, the Application shall have its authentication immediately revoked.
6) Revocation can occur anytime by posting the <Application AIM ID, Token Value>, to a revocation list.
7) Each instance of an application shall be unique. Thus, each instance shall have its own unique token.

[1] The File Asset Tag, as defined in Slide 3, shall be used as the Application Asset Tag.

BACKGROUND INFORMATION

keyed-Hash Message Authentication Code[1]

HMAC can provide authentication using a shared secret instead of using digital signatures with asymmetric cryptography. It trades off the need for a complex public key infrastructure by delegating the key exchange to the communicating parties, who are responsible for establishing and using a trusted channel to agree on the key prior to communication.

This definition is taken from RFC 2104:

$$\mathrm{HMAC}(K, m) = \mathrm{H}\left((K' \oplus opad) \parallel \mathrm{H}\left((K' \oplus ipad) \parallel m \right) \right)$$

$$K' = \begin{cases} \mathrm{H}(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

where

H is a cryptographic hash function
m is the message to be authenticated
K is the secret key
K' is a block-sized key derived from the secret key, K; either by padding to the right with 0s up to the block size or by hashing down to less than or equal to the block size first and then padding to the right with zeros.
|| denotes concatenation.
⊕ denotes bitwise exclusive or (XOR).
opad is the block-sized outer padding, consisting of repeated bytes valued 0x5C.
ipad is the block-sized inner padding, consisting of repeated bytes valued 0x36[2].

1. https://en.wikipedia.org/wiki/HMAC#cite_note-:0-3
2. *"Definition of HMAC". HMAC: Keyed-Hashing for Message Authentication. sec. 2. doi:10.17487/RFC2104. RFC 2104.*

The Fisher-Yates Shuffle

The Fisher-Yates Shuffle is a methodology used by AIM to randomly select parameters from the Parameter Data Vector (PDV), for which those parameters are then used to create new security credentials.

The following is taken directly from: "Fisher-Yates Shuffle" Wikipedia Link

The **Fisher–Yates shuffle** is an algorithm for shuffling a finite sequence. The algorithm takes a list of all the elements of the sequence, and continually determines the next element in the shuffled sequence by randomly drawing an element from the list until no elements remain.[1] The algorithm produces an unbiased permutation: every permutation is equally likely. The modern version of the algorithm takes time proportional to the number of items being shuffled and shuffles them in place.

The Fisher–Yates shuffle is named after Ronald Fisher and Frank Yates, who first described it. It is also known as the **Knuth shuffle** after Donald Knuth.[2] A variant of the Fisher–Yates shuffle, known as **Sattolo's algorithm**, may be used to generate random cyclic permutations of length *n* instead of random permutations.

1) Eberl, Manuel (2016). "Fisher–Yates shuffle". Archive of Formal Proofs. Retrieved 28 September 2023.
2) Smith, James (2023-04-02). "Let's Do the Knuth Shuffle". Golang Project Structure. Retrieved 2023-04-03.

The Fisher-Yates Shuffle (Original Method)

The Fisher–Yates shuffle, in its original form, was described in 1938 by Ronald Fisher and Frank Yates in their book *Statistical tables for biological, agricultural and medical research*.[3] Their description of the algorithm used pencil and paper; a table of random numbers provided the randomness. The basic method given for generating a random permutation of the numbers 1 through *N* goes as follows:

1) Write down the numbers from 1 through *N*.
2) Pick a random number *k* between one and the number of unstruck numbers remaining (inclusive).
3) Counting from the low end, strike out the *k*th number not yet struck out, and write it down at the end of a separate list.
4) Repeat from step 2 until all the numbers have been struck out.
5) The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

Provided that the random numbers picked in step 2 above are truly random and unbiased, so will be the resulting permutation. Fisher and Yates took care to describe how to obtain such random numbers in any desired range from the supplied tables in a manner which avoids any bias. They also suggested the possibility of using a simpler method — picking random numbers from one to *N* and discarding any duplicates—to generate the first half of the permutation, and only applying the more complex algorithm to the remaining half, where picking a duplicate number would otherwise become frustratingly common.

3) *Fisher, Ronald A.; Yates, Frank (1948) [1938]. Statistical tables for biological, agricultural and medical research (3rd ed.). London: Oliver & Boyd. pp. 26–27. OCLC 14222135.* Note: the 6th edition, ISBN 0-02-844720-4, is available on the web, but gives a different shuffling algorithm by C. R. Rao.

The Fisher-Yates Shuffle (Modern Version)

The modern version of the Fisher–Yates shuffle, designed for computer use, was introduced by Richard Durstenfeld in 1964[4] and popularized by Donald E. Knuth in *The Art of Computer Programming* as "Algorithm P (Shuffling)".[5] Neither Durstenfeld's article nor Knuth's first edition of *The Art of Computer Programming* acknowledged the work of Fisher and Yates; they may not have been aware of it.[citation needed] Subsequent editions of Knuth's *The Art of Computer Programming* mention Fisher and Yates' contribution.[6]

The algorithm described by Durstenfeld is more efficient than that given by Fisher and Yates: whereas a naïve computer implementation of Fisher and Yates' method would spend needless time counting the remaining numbers in step 3 above, Durstenfeld's solution is to move the "struck" numbers to the end of the list by swapping them with the last unstruck number at each iteration. This reduces the algorithm's time complexity to compared to for the naïve implementation.[7] This change gives the following algorithm (for a zero-based array).

```
-- To shuffle an array a of n elements (indices 0..n-1):

for i from n−1 down to 1 do
        j ← random integer such that 0 ≤ j ≤ I
        exchange a[j] and a[i]
```

An equivalent version which shuffles the array in the opposite direction (from lowest index to highest) is:

```
-- To shuffle an array a of n elements (indices 0..n-1):

 for i from 0 to n−2 do
        j ← random integer such that i ≤ j < n
        exchange a[i] and a[j]
```

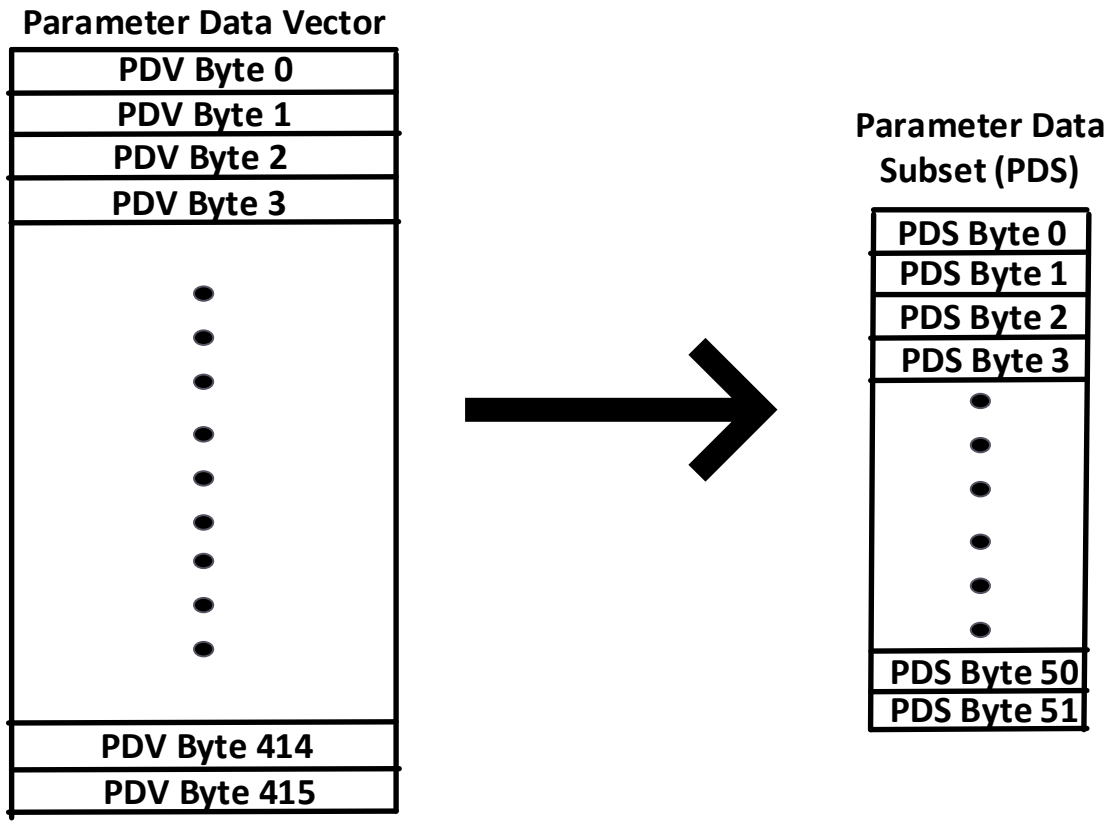
4) Durstenfeld, R. (July 1964). "Algorithm 235: Random permutation" (PDF). Communications of the ACM 7 (7). 420. doi:10.1145/364520.364540. S2CID 494994
5) Knuth, Donald E. (1969). Seminumerical algorithms. The Art of Computer Programming. Vol. 2. Reading, MA: Addison–Wesley. pp. 139–140. OCLC 85975465.
6) Jump up to: a b Knuth (1998). Seminumerical algorithms. The Art of Computer Programming. Vol. 2 (3rd ed.). Boston: Addison–Wesley. pp. 12–15, 145–146. ISBN 0-201-89684-2. OCLC 38207978.
7) Black, Paul E. (2005-12-19). "Fisher–Yates shuffle". Dictionary of Algorithms and Data Structures. National Institute of Standards and Technology. Retrieved 2007-08-09.

PDV and PDS Details

- The Parameter Data Value (PDV) is a vector that is an array of bytes that by design is a multiple of the size of the Parameter Data Set. Ideally, the multiple should be no less than eight (8).
- The Parameter Data Set (PDS) is a vector that is an array of bytes and is used to hold the values that will become the next session's encryption key, HMAC secret and seed values. Thus, the PDS size is the sum of the following entities:
  - Size of the Bulk Encryption Key
  - Size of the HMAC Key
  - Size of the Session Seed
- The default size for these values are 16-bytes (i.e., AES128) for the bulk encryption key, 32-bytes for the HMAC secret (i.e., SHA256 based HMAC), and 4-bytes for the session seed value, thus making it a total of 52-bytes. If the PDV is the default multiple of eight (8) for a PDS of sized, 52-bytes, then the PDV will be of size, 416-bytes.

Parameter Data Vector
PDV Byte 0
PDV Byte 1
PDV Byte 2
PDV Byte 3
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
PDV Byte n-3
PDV Byte n-2
PDV Byte n-1

Derivation of SDS from PDV



In going from the PDV to creation of the SDS, the AIM framework must first select the Parameter Data Subset (PDS) from which the parameters will be used to create Next Session Credentials.