Minimally-Configured Security Engine
(Automatic Implementation of Secure Silicon)
University of Florida

Release 1.1
May 8, 2024

Acknowledgements

... *To be written* ...

Contents

Chapter 1

Introduction and Simulation Top-level

1.1 Introduction

Purpose of this document/target audience(s)

This document[1] is intended for people trying to use the `AISS` configuration of the Minimally-Configured Security Engine `(MCSE)`, and for people trying to understand the architecture and security primitives integrated with `MCSE` for providing security assurance as per the `AISS Phase III` program goals and metrics. It is intended to provide top-level context for the code and code structure so that you can navigate the directories, read the code, and/or understand how to use this `MCSE` configuration.

This document does not attempt to do a detailed code explanation of the `MCSE` code in the repository. Rather, this document is only meant to provide you with high-level context and structure so that you can, on your own, easily navigate through the directories and files.

1.2 Common Directory Organization

The `MCSE` repository is organized as per the following structure:

- The directory `source_RTL/` contains `mcse` (in file `mcse_top.sv`) and everything inside it, including the boot control, cryptographic blocks (SHA 256 and Camellia 256), PUF Control Module, scan protection, lifecycle management module, firmware authentication module, and all the required interfaces for integration with the `TA2` SoC.

- The directory `tb/` contains top-level testbench `mcse_top_tb.sv` and other verification collateral, makefiles, etc.

- The `synthesis/` contains copies of a few Verilog library files `(LEDA)` used by `MCSE` and can be treated as black boxes. It also contains the synthesis scripts for the GF12 library.

[1]This document should be considered living, where updates and revisions are expected.

Some Important RTL Source Code Hierarchy and Description

```
MCSE Top
│
├── Minimum Security Module
│   ├── SHA-256
│   ├── Camellia-256
│   ├── Boot Control Interface (GPIO)
│   ├── PCM
│   ├── AHB-Lite Interface
│   └── AHB-Lite Translation Module
└── MCSE Control Unit
    ├── Life-cycle Protection Module
    ├── Secure Memory
    ├── Secure Boot Control
    ├── Firmware Authentication Module
    └── Scan Protection Module
```

1. **mcse_top.sv:** This is the top module for `MCSE`. The I/O contains the gpio_out, gpio_in, system-side AHB requester ports, and abstracted ports for lifecycle authentication and transitions which will later be done through the VimScan TAP ports. This module instantiates the minimum security module and the `MCSE` control unit.

2. **min_security_module.sv:** This contains the cryptographic accelerators (SHA-256 and CAM-256) for hashing and encryption/decryption. The GPIO module and PCM are instantiated here as well. This also houses the AHB-lite bus interface and VimScan protection modules.

3. **lifecycle_protection.sv:** This is the life-cycle protection module that holds the life-cycle state and handles requests for life-cycle transitions. This module also instantiates the memory in `lc_memory.sv` (modeled as a read-only register file) which contains the keys to transition the life-cycle state for each life-cycle.

4. **secure_memory.sv:** This instantiates the secure memory modeled as a read-and-write register file. This memory will house the 2048 asset bits, life-cycle authentication keys, and anything miscellaneous `MCSE` needs to store.

5. **secure_boot_control.sv:** This instantiates the secure boot control module which is the core of `MCSE`. This module is responsible for controlling all other modules in `MCSE`. Will handle the control flow for handshaking with TA2 (TA2 SoC reset, TA2 system bus wakeup, IP ID Extraction, TA2 normal operations release, FW authentication), life-cycle authentication and transition, and AHB-lite bus interface control.

1.3 Simulation and Synthesis

1.3.1 `MCSE` RTL Simulation

The current flow used Synopsys VCS in the `AISS Cloud` for all simulation experiments.

The pre-synthesis testbench is `mcse_top_tb.sv`. To run it, use the command below (currently has to be run in the `src_RTL/` directory but will be fixed)

```
   $ make MCSEtest
```

The testbench for a gate-level netlist (post-synthesis) simulation is `mcse_top_netlist_tb.sv`. Run the command below for a gate-level simulation:

```
   $ make NETLISTtest
```

1.3.2 `MCSE` Synthesis on GlobalFoundries GF12LPP

The current flow used Synopsys Design Compiler in the `AISS Cloud` for all synthesis experiments.

For synthesis, the `compiledc.tcl` file is used, and a gate-level netlist file `mcse_netlist.v` is produced. Run the command below:

```
   $ make synthesis
```

Simulation and Synthesis Source File Description

- **mcse_top_tb.sv** is the pre-synthesis testbench. This simulation covers the first boot and subsequent boot functionality for each lifecycle state with descriptive output messages. Each task serves to test `MCSE` life-cycle-specific functionality in a modular fashion. It is a behavioral simulation and walks through each life-cycle sequentially as if to cover all functionality throughout the chip's life.

- **mcse_top_netlist_tb.sv** is the post-synthesis (gate level netlist) testbench. This simulation highlights the same functionalities as in the pre-synthesis simulation but doesn't have any output messages to show internal registers. This simulation shows first boot functionality for all lifecycles except for end-of-life. The focus here is to show that `MCSE`/TA2 interactions are still functional post-synthesis.

- **Makefile:** will clean the work and build environments for simulation and synthesis. All previously generated temporary files are removed such that any new run of simulation or synthesis is clean. Please refer above for the proper commands to execute each simulation and synthesis of `MCSE`.

- **compiledc-GF12.tcl:** is the Synopsys Design Compiler script to synthesize `MCSE`. The script allows DC to perform heavy optimizations to help reduce gate count. Once synthesis is finished, the script will display the netlist area, hierarchy, and references, and then output the gate-level netlist to a Verilog file which can then be used in post-synthesis simulation.

Chapter 2

MCSE Architecture

2.1 High-level Architecture

The MCSE architecture (Fig. 2.1) configuration in this document complies with the DARPA metrics for AISS Phase III. To meet the gate count constraint, the configuration shown here is kept at a minimum, and comprises the following:

- **Secure boot control and boot interface:** This module acts as the "security-brain" of the MCSE and controls all operations during boot and all security countermeasure enforcement in the SoC.
- **Lifecycle Management Module:** This module is responsible for end-to-end lifecycle management of the device from fabrication to end-of-life. It ensures differential access to assets, privilege levels, etc. based on the lifecycle of the device.
- **Firmware Authentication Module:** This module checks the authenticity of the firmware image and its signature before allowing the TA2 to deploy the firmware image.
- **Scan Protection Module:** This ensures that the scan-chain is protected from unauthorized access.
- **Crypto Modules:** These modules[1] comprise a SHA-256 and a Camellia-256 block cipher to enable crypto operations within MCSE, ensuring the security of assets when they are stored on-chip.
- **PUF Control Module (PCM):** The PCM carries out error correction on the MeLPUF signatures generated from all the MeLPUF instances in both the MCSE and the TA2 SoC.

Security primitives are embedded throughout the MCSE architecture based on the threat model of the AISS program. MCSE enforces security via control and coordination of countermeasures using the architectural components highlighted above.

The interfaces provided with MCSE include:

- Boot interface using GPIO (Table 2.1)
- AMBA AHB-Lite bus interface (Table 2.2)
- VimSCAN access ports interface(Table 2.3)

[1]Both the SHA-256 and the Camellia-256 contain 256-bit MeLPUF instances to generate the MCSE Si ID.

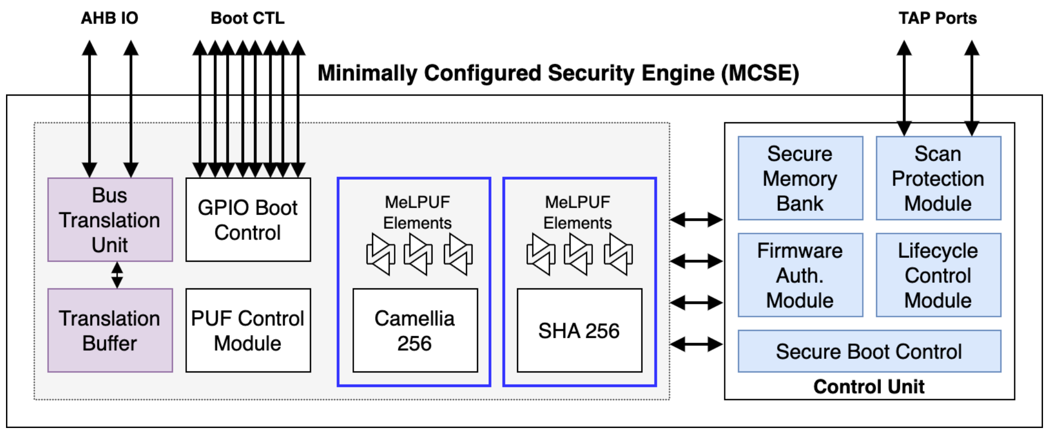

Figure 2.1: MCSE Architecture for AISS Phase III.

2.2 IO Pin Mapping

2.2.1 MCSE/TA2 Boot Interface Pin Mapping

GPIO Pin	Direction	Functionality	GPIO Pin	Direction	Functionality
0	Output	Reset TA2 SoC	1	Input	TA2 Reset ACK
2	Output	Halt TA2 SoC	3	Input	TA2 Halt ACK
4	Output	Normal Operation Release	5	Input	TA2 Norm. Op. ACK
6	Output	TA2 Bus Wakeup	7	Input	TA2 Bus Wakeup ACK
8	Output	IPID Address[0]	–	–	–
9	Output	IPID Address[1]	–	–	–
10	Output	IPID Address[2]	–	–	–
11	Output	IPID Address[3]	–	–	–
12	Output	IPID Trigger	13	Input	IPID Valid
15	Output	FW Auth Success ACK	14	Input	FW Auth. Request
16	Output	FW Auth Failure ACK	–	–	–

Table 2.1: MCSE/TA2 Boot Control Interface Pin Mapping (each pin is 1 bit wide).

2.2.2 MCSE/TA2 Bus Interface Pin Mapping

The Bus Interface Pin Mapping is provided in Table 2.2.

2.2.3 VimSCAN Test Access Ports Pin Mapping.

The VimSCAN TAP Pin Mapping is provided in Table 2.3.

IO Name	Width	Direction
I_hrdata	1-bit	Input
I_hready	1-bit	Input
I_hresp	1-bit	Input
I_hreadyout	32-bit	Input
O_haddr	1-bit	Output
O_hburst	1-bit	Output
O_hmastlock	1-bit	Output
O_hprot	1-bit	Output
O_hnonsec	1-bit	Output
O_hsize	1-bit	Output
O_htrans	1-bit	Output
O_hwdata	1-bit	Output
O_hwrite	1-bit	Output

Table 2.2: AHB-Lite Bus Interface Pin Mapping (see protocol specification here).

IO Name	Width	Direction	Functionality
scan_in	1-bit	Input	Serial input to VimSCAN
tc	1-bit	Input	0: Normal Mode, 1: VimSCAN Mode
vim_en	1-bit	Input	Control for VimSCAN register shift
vim_in[2]	32-bit	Input	Parallel input to VimSCAN for observability
scan_out	1-bit	Output	Serial output from VimSCAN

Table 2.3: VimSCAN Interface Pin Mapping.

2.3 `MCSE` Boot Control Sequence

For this instance of `MCSE` for supply-chain, the boot control consists of a 4-stage sequence between itself and the TA2 SoC, as shown in Fig. 2.2. This boot control design standardizes the interaction between `MCSE` and the TA2, which divides their responsibility during system boot. While `MCSE` is responsible for *only* the security aspect, the TA2 is responsible for the SoC wake-up and boot. – However, `MCSE` controls the initiation of TA2 system boot after successfully booting up its security subsystem, as shown in Fig. 2.2.

2.3.1 High-Level `MCSE` Lifecycle Functionality

- **Manufacture and Test Lifecycle**
  `MCSE` Initialization→Reset TA2→Golden ChipID Generation*[3]→Update Lifecycle→ →Transition Lifecycle
- **Packaging and OEM Lifecycle**
  `MCSE` Initialization→Reset TA2 SoC→Lifecycle Auth.*→Challenge Chip ID Gener-

[3]All operations marked * are only performed at the first boot of each lifecycle.

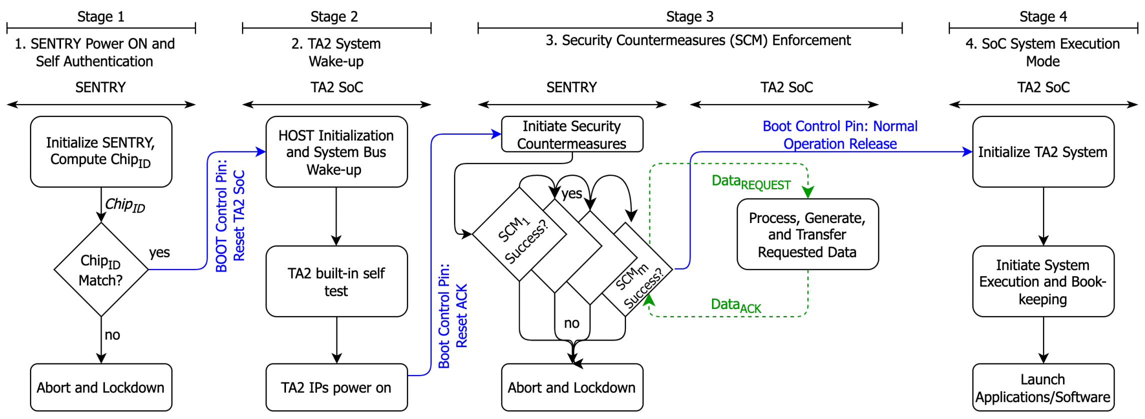
Stage 1
1. SENTRY Power ON and Self Authentication
SENTRY
Initialize SENTRY, Compute Chip_ID
Chip_ID
Chip_ID Match?
yes
no
Abort and Lockdown
BOOT Control Pin: Reset TA2 SoC
Stage 2
2. TA2 System Wake-up
TA2 SoC
HOST Initialization and System Bus Wake-up
TA2 built-in self test
TA2 IPs power on
Boot Control Pin: Reset ACK
Stage 3
3. Security Countermeasures (SCM) Enforcement
SENTRY
TA2 SoC
Initiate Security Countermeasures
SCM1 Success?
yes
SCMm Success?
no
Abort and Lockdown
Boot Control Pin: Normal Operation Release
DataREQUEST
Process, Generate, and Transfer Requested Data
DataACK
Stage 4
4. SoC System Execution Mode
TA2 SoC
Initialize TA2 System
Initiate System Execution and Book-keeping
Launch Applications/Software

Figure 2.2: Boot interaction flow diagram between MCSE and TA2 SoC via the Boot Control Interface for a Supply-chain use-case. Only communication between MCSE and TA2 is shown.

ation* →Chip ID Authentication*→Normal Operation Release to TA2→Firmware Authentication→Update Lifecycle→Transition Lifecycle

- **Deployment Lifecycle**
  MCSE Initialization→Reset TA2 SoC→Lifecycle Auth.*→Challenge Chip ID Generation* →Chip ID Authentication*→Normal Operation Release to TA2→Update Lifecycle →Transition Lifecycle

- **Recall Lifecycle**
  MCSE Initialization→Reset TA2 SoC→Lifecycle Auth.*→Challenge Chip ID Generation* →Chip ID Authentication*→Purge Process→Update Lifecycle→Transition Lifecycle

- **End-of-Life Lifecycle**
  MCSE Initialization→Reset TA2 SoC→Lifecycle Auth.*→Truncated Boot Sequence

MCSE provides architectural features to simplify the overall boot process (shown in Fig. 2.2) by decoupling the actions of the TA2 CPU to wake up different system components from the security-related actions such as unlocking and authentication of the constituent IPs. A key requirement in enforcing security at boot is to enable a secure transfer of data (assets including PUF signatures, watermarks, cryptographic keys, etc.) from the TA2 to MCSE. An interesting conundrum for boot is the retrieval of data from the TA2 to MCSE. Since the main system bus is inactive in the first phase of boot, MCSE issues an interrupt[4] using the Boot Control Interface, to the TA2 for initialization and system bus wake-up. This is followed by an acknowledgment from the TA2 to MCSE indicating successful Phase 2 operations. At this stage of the boot, the system bus is active and security can be enforced. Security countermeasures can be executed in chaining or independent of preceding countermeasures, contingent upon the nature of the countermeasures or the devised mitigation strategy. After all successful countermeasure executions, MCSE relinquishes system access to TA2 via an interrupt, after which the SoC can initiate system operation and launch applications.

[4]All interrupts are issued using the Boot Control Interface, and their pin mappings and functionalities are provided in Table 2.1.

Note: We briefly explain the security functionalities like the authentication protocols, life-cycle transitions, etc. in subsequent chapters in this document.

Chapter 3

Supply-Chain Assets

MCSE uses specific classes of assets for the mitigation of threats defined in the AISS threat model. The exhaustive asset class and description is as follows:

MCSE ID: 256 bits

1. This is the MCSE ID derived from MeLPUF instances inside the Camellia 256 and SHA 256 blocks in the MCSE.
2. Corresponding threat vectors include:
   - Device counterfeiting and over-manufacturing
   - Device reuse and tracking

Composite IPID for TA2 IPs: 256 bits

1. This is the IPID derived from MeLPUF instances inside IPs in the TA2 SoC.
2. Corresponding threat vectors include:
   - IP piracy and over-manufacturing
   - IP reuse

See Chapter 5 for uses of these assets in the authentication protocols.

Firmware Signing Key: 256 bits

1. This is the golden firmware signing key shared across all legal and affiliated parties authorized to issue firmware upgrade requests to MCSE. This version of MCSE uses an extended version of the HMAC signing protocol[1] to sign and issue/share firmware binaries (see Chapter 6 for the firmware signing and authentication process).
2. Corresponding threat vectors include:

[1]HMAC (Hash-Based Message Authentication Codes) Definition, See Okta HMAC.

- Firmware integrity attacks
- Firmware confidentiality attacks

Scan Protection Key: 256 bits

1. MCSE uses a key to protect scan access from unauthorized parties (see Chapter 7 for Scan protection).
2. The Scan protection is built on the low-overhead scan-protection VimScan[2] architecture.
3. Corresponding threat vectors include:
    - Scan-based controllability attacks
    - Scan-based observability attacks

Secure Communication Key: 256 bits

1. MCSE uses this key for encryption to protect all assets and data transfer outside its boundary to ensure the confidentiality of the asset.
2. Corresponding threat vectors include:
    - Asset leakage
    - Improper provisioning

Composite Watermark: 256 bits

1. MCSE uses watermarks as an additional layer of IP authentication in the TA2 platform (see Chapter 5 for more details).
2. The watermark protection is built on the SIGNED[3] watermark model.
3. Corresponding threat vectors include:
    - IP piracy
    - Malicious modifications of IPs in an SoC

Other supply-chain identification assets include:

- **Manufacturer ID: 128 bits**
- **System Integrator ID: 128 bits**

The total asset size of the MCSE platform for Phase III is `2048 bits`.

[2]VIm-Scan: A Low Overhead Scan Design Approach for Protection of Secret Key in Scan-Based Secure Chips, see here.

[3]SIGNED: A Challenge-Response Scheme for Electronic Hardware Watermarking, see here.

Chapter 4

Lifecycle Management

The SoC transitions through various pre-designated lifecycles throughout its lifespan, starting from the time it was fabricated onto a die to the time when the chip is decommissioned from operation. There are many ways to map the lifecycles to different parties depending on the scenario. One example mapping to consider is an Original Equipment Manufacturer (OEM) buying parts straight from a manufacturer, integrating them into a single system, and then leasing the system to end users. The reasoning behind the choice of specific lifecycles is informed by expected state transitions which include the following activities:

- Secure provisioning of identities (IDs) and firmware.
- Verification of firmware integrity at boot.
- Attestation of system hardware and software integrity.
- Update of IDs
- Mutual authentication of components within a system or a package
- Supply chain tracking including device dis-enrollment, and optionally re-enrollment.

Each lifecycle has its own set of security and non-security operations, data (including assets) generation, movement, and processing which enable the security primitives to be imposed

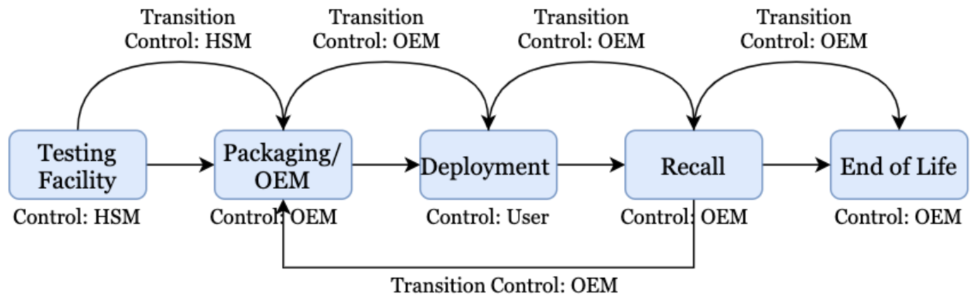

Figure 4.1: SoC Lifecycle and Transition Control Sequence

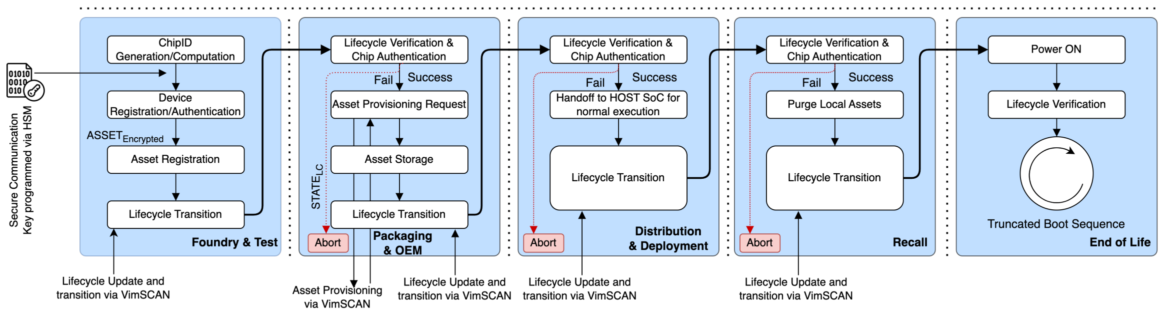
Secure Communication Key programmed via HSM
ChipID Generation/Computation
Device Registration/Authentication
ASSET_Encrypted
Asset Registration
Lifecycle Transition
Foundry & Test
Lifecycle Update and transition via VimSCAN
Lifecycle Verification & Chip Authentication
Fail
Success
Asset Provisioning Request
Asset Storage
STATE_LC
Lifecycle Transition
Abort
Packaging & OEM
Asset Provisioning via VimSCAN
Lifecycle Update and transition via VimSCAN
Lifecycle Verification & Chip Authentication
Fail
Success
Handoff to HOST SoC for normal execution
Lifecycle Transition
Abort
Distribution & Deployment
Lifecycle Update and transition via VimSCAN
Lifecycle Verification & Chip Authentication
Fail
Success
Purge Local Assets
Lifecycle Transition
Abort
Recall
Lifecycle Update and transition via VimSCAN
Power ON
Lifecycle Verification
Truncated Boot Sequence
End of Life

Figure 4.2: Life-cycle management using MCSE's boot control sequence.

by SENTRY. This document goes into detail about what such operations are and how these operations are orchestrated. Lifecycle state transitions (Fig. 4.1) are possible only when the authorized entity is requesting a transition, and the requested transition has to be a valid transition. The flow of operations in each lifecycle is described in Fig. 4.2.

4.1 Manufacture and Test

This lifecycle represents the steps between when chips are fabricated, tested, and accepted (manufactured) and when chips receive their first set of unique identifiers (ChipID). Within AISS, this step is assumed trusted[1]. For minimal security, a device ID (ChipID) that uniquely identifies the specific device is expected. During this step, ChipID is generated and registered. After successful registration of the ChipID, other assets such as the lifecycle state, PUF CRPs, communication keys, and other relevant assets are registered with the AMI (supply chain tracking database).

Location, Control, and Assets

Relevant information during this lifecycle includes the following:

- Location of the chip: Testing Facility
- Entity in control of the chip: HSM
- Assets[2]: ChipID, Communication Key, Composite IPIDs, currOwnerID, Manufacturer ID (see Table 4.1).

[1]ChipID and other assets in this lifecycle require the assistance of a Hardware Security Module (HSM) and MCSE, which mediates the interaction between AMI and MCSE for registration. We assume the presence of such an intermediary during operation.

[2]Each asset except the communication key is encrypted before on-chip storage.

Asset	Access Level	Storage (on-chip)
Composite IPID	Read-only	Yes
ChipID	Read-only	Yes
Communication Key	Read-only	Yes
currOwnerID	Read-only	Yes
Manufacturer ID	Read-only	Yes
Lifecycle State	Read/Write	Yes

Table 4.1: Asset Permission Levels and Storage in the Manufacture and Test Lifecycle.

4.2 Packaging/OEM

Chips may be integrated after fabrication/testing as an SoC or system in a package. The transition from provisioned to integrated is to establish a unique integrated identity. This might include the ability to store additional assets (such as IDs, firmware signatures, etc.). Furthermore, at the minimum, an algorithm for mutual authentication between the chip and system integrator as well as between chips must also be included. This is included as part of the `MCSE`. In this lifecycle, the only communication that happens between `MCSE` and the TA2 platform is the handshake and transfer of IPID signatures.

Location, Control, and Assets

Relevant information during this lifecycle includes the following:

- Location of the chip: Packaging/OEM
- Entity in control of the chip: OEM
- Assets: ChipID, Communication Key, Composite IPIDs, currOwnerID, Firmware Signing Key, Manufacturer ID, System Integrator ID (see Table 4.2)

Asset	Access Level	Storage (on-chip)
Composite IPID	Read-only	Yes
ChipID	Read-only	Yes
Communication Key	Read-only	Yes
currOwnerID	Read-only	Yes
Firmware Signing Key	Read-only	Yes
Manufacturer ID	Read-only	Yes
System Integrator ID	Read-only	Yes

Table 4.2: Asset Permission Levels and Storage in the Packaging/OEM Lifecycle.

4.3 Deployment

Location, Control, and Assets

Relevant information during this lifecycle includes the following:

- Location of the chip: In-field
- Entity in control of the chip: End user
- Assets: ChipID, Communication Key, Composite IPIDs, currOwnerID, Firmware Signing Key, Manufacturer ID, System Integrator ID (see Table 4.3)

Asset	Access Level	Storage (on-chip)
Composite IPID	Read-only	Yes
ChipID	Read-only	Yes
Communication Key	Read-only	Yes
currOwnerID	Read-only	Yes
Firmware Signing Key	Read-only	Yes
Manufacturer ID	Read-only	Yes
System Integrator ID	Read-only	Yes

Table 4.3: Asset Permission Levels and Storage in the Deployment Lifecycle.

4.4 Recall

The core idea is to ensure that the only long-term end-user changes to the chip or device are 1) physical changes due to use and 2) manufacturer or integrator firmware upgrades. Before any operations are performed during the recall lifecycle, it is critical to note that the device transitions from deployment to recall under the control of the OEM. Hence, it is crucial to purge any end-user assets, firmware, and application code from the device to avoid unauthorized user data asset access requests.

At this stage, the OEM can decide on whether to re-enroll the device or decommission the device entirely. Some use cases may allow a device to re-enter the supply chain after dis-enrollment, for example through authorized device recycling or refurbishment channels. In these cases, steps for the âOEM Lifecycleâ are repeated. After a device is disenrolled, it might also get disposed of. Disposal ensures that all data, whether it is from the manufacturer, integrator, or end-user is removed from the device. It is important to note that this transition might be skipped entirely. For example, a smartphone could be run over by a bus, which means it is effectively âdisposed ofâ despite not going through the transitions. Similar to how the failed update transition is not depicted in the diagram, this (and others) are not depicted but still should be considered.

Location, Control, and Assets

Relevant information during this lifecycle includes the following:

- Location of the chip: OEM
- Entity in control of the chip: OEM
- Assets: ChipID, Communication Key, Composite IPIDs, currOwnerID, Manufacturer ID, System Integrator ID (see Table 4.4)

Asset	Access Level	Storage (on-chip)
Composite IPID	Read-only	Yes
ChipID	Read-only	Yes
Communication Key	Read-only	Yes
currOwnerID	Read-only	Yes
Manufacturer ID	Read-only	Yes
System Integrator ID	Read-only	Yes

Table 4.4: Asset Permission Levels and Storage in the Recall Lifecycle.

4.5 End-of-Life

In this lifecycle, no operations are allowed to happen. Once the OEM transitions the device lifecycle to `end-of-life`, all assets including the ChipID, communication keys, IPIDs, etc. are erased from the `MCSE` secure memory, and the device enters a truncated boot sequence. A device in this lifecycle is considered dead and all data (and metadata) associated with it are erased.

Chapter 5

Authentication Protocols

`MCSE` uses multiple authentication checks to authenticate various components and aspects of the SoC. These include:

- Composite IPID Authentication
- ChipID Authentication
- Lifecycle Authentication and Transition
- Other Identity Authentication

To carry out the Composite IPID and ChipID authentication checks, `MCSE` uses signatures derived from `MeLPUF`[1].

`MeLPUF` exploits the observation that every chip die has a unique doping density and these variations at the sub-micron level result in different capacitance values in the uninitialized bi-stable memory cells. The variations result in the extracted value from these being either `1` or `0`. By integrating the cells in specific IPs/regions of the SoC, `MeLPUF` can generate a unique, unclonable ID for the IP which can act as a watermark; furthermore, by distributing MeLPUF cells across multiple IPs in the SoC, it is possible to collect a signature of the entire SoC that can act as a ChipID. Each `MeLPUF` bit incurs an overhead of 6 NAND gates.

5.1 Composite IPID Authentication

Select IPs in the SoC may be modified to include $\mathcal{N}$ `MeLPUF` elements. These include IPs subject to threat vectors including IP piracy and malicious modifications (also discussed in Chapter 3). `MCSE` authenticates these IPs during secure boot and uses these IPIDs to generate the ChipID for the SoC (discussed in the next section). We now explain the composite IPID generation and authentication.

Note: This authentication *only* happens at the first boot of *any* device lifecycle.

[1]MeLPUF: Memory-in-Logic PUF Structures for Low-Overhead IC Authentication, see here.

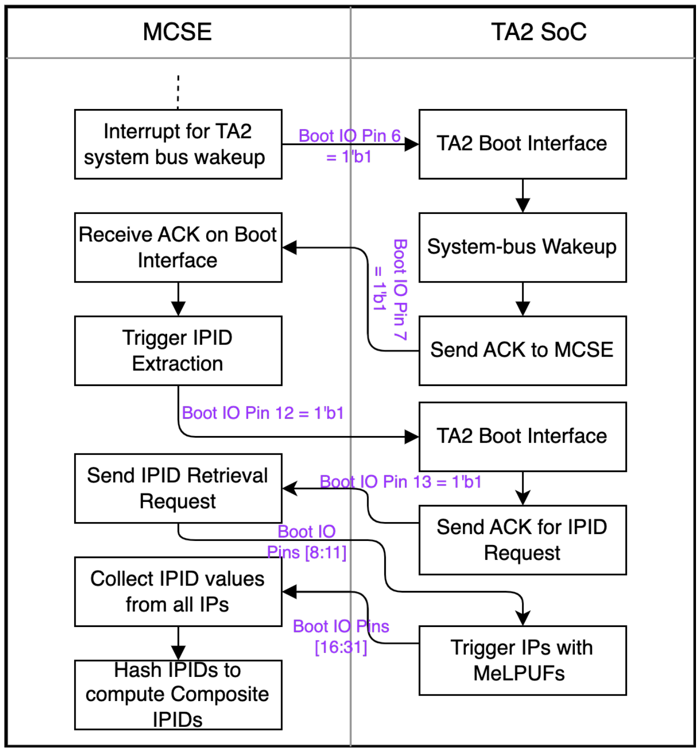

Figure 5.1: Composite IPID Generation from Golden IPIDs.

Composite IPID Generation

For `MCSE` to authenticate these IPIDs, it first needs to generate the composite IPIDs from the golden IPID values. This is done in the first boot of the "manufacture and test" lifecycle (see Fig. 5.1). The following sequence of steps is followed:

- Select IPs in the TA2 SoC are augmented with the security wrapper. These IP blocks have an embedded $\mathcal{N}$-bits `MeLPUF` to generate unique PUF signatures.
- At power on, the `MCSE` triggers `(BOOT IO pin 6)` the TA2 to wake up the system bus, and upon receiving an acknowledgment `(BOOT IO pin 7)` for the same, triggers the TA2 to initiate the PUF signature value transfer from $\mathcal{N}$ `MeLPUF` elements distributed across IPs in the TA2.
- `MCSE` triggers[2] each TA2 IP with `MeLPUF`, and receives the signature over the Boot interface[3].
- Each IPID value is collected, concatenated, and hashed using SHA-256, and the composite IPID value is derived.

$$\text{Composite IPID} = [\text{Concatenate}(\text{IPID}_1,\text{IPID}_2,\text{IPID}_3.....\text{IPID}_\text{n})]_\text{HASH}$$

[2] `MCSE` uses `(BOOT IO pin 8:11)` to map IPs in TA2, and as such, supports `MeLPUF` signature retrieval from 16 IPs in the TA2.

[3] The collection of `MeLPUF` signatures in the alpha release was done using `(BOOT IO pin 16:31)` but this will be migrated to the AHB-Lite bus interface.

Note: The above sequence of operations when performed for the first time at virgin-boot is used to register the golden values of composite IPIDs. These operations are also repeated each time `MCSE` needs to verify different IDs, but those generated values are considered challenge responses.

Authentication

- At this point, the golden composite IPID has already been generated.
- The above-described sequence for generating golden IPIDs is repeated to generate the challenge response IPIDs.
- `MCSE` compares the challenge response to the golden response of each IP to verify the authenticity of each IP.

5.2 ChipID Authentication

ChipID used in `MCSE` is a 256-bit asset, comprising two parts: (1) Composite IPID and (2) MCSE Si ID. We already discussed how the Composite IPID is generated. The MCSE Si ID is generated by `xoring` the `MeLPUF` signatures of the SHA-256 and the Camellia-256 inside `MCSE`.

Before generating and verifying the Composite IPID, the `MCSE` first verifies its own MCSE Si ID. The golden value of MCSE Si ID is also generated before the generation of the golden Composite IPID.

ChipID Generation: The ChipID is created using the following sequence:

$$\text{ChipID} = \text{MCSE Si ID} \oplus \text{Composite IPID}_{\text{HASH}}$$

Authentication

At first power on in each lifecycle, `MCSE` verifies the SoC by authenticating the ChipID. To verify the ChipID, the following sequence of steps is followed:

- `MCSE` first computes the MCSE Si ID and then the Composite IPID. The steps involved in this have been explained.
- Then, `MCSE` compares these challenge-response pairs to the golden value of ChipID and depending on the outcome, determines the authenticity of the SoC.
- If the authentication is successful, `MCSE` schedules the next sequence of operations (see Fig. 4.2).

5.3 Lifecycle Transition and Authentication

Lifecycle management in `MCSE` enables differential access to certain assets stored in the secure on-chip memory (see Chapter 3). As such, `MCSE` protects lifecycle transitions using certain identifiers to allow only valid parties to transition to authorized and permittable lifecycles. `MCSE` uses two identifiers, `currOwnerID` and `TransitionID`.

- `currOwnerID`: Identifies whether a party is authorized to boot into the current lifecycle state of the device (see Table 5.2 and Algorithm 2).
- `TransitionID`: Identifies whether a party is authorized to transition the device into the next lifecycle (see Table 5.1 and Algorithms 1).

Lifecycle	**TransitionID (256 bits)**
Manufacture & Test	h33a344a35afd82155e5a6ef2d092085d704dc70561dde45d27962d79ea56a24a
Packaging & OEM	h988b6a57b75f5696f01b8207b1c99bc888b4a2421a0ab4b29bd302f5b8a93348
Deployment	h4893565d146d9fa19dc850e0c409b2a62ec5cb53eea4d4719c93a882f988284e
Recall	hcabc36e4f52fcd1a8b62d82d975e4c8595da7f6df52e2143174c3dc8b3870e03
End-of-Life	hc3e0fed656de0ab97d4c2e2f8798ff16c8c4ac54192046fc72debd8e4fd801e5

Table 5.1: `Transition ID` for Different Lifecycles.

Lifecycle	**currOwnerID (256 bits)**
Manufacture & Test	h431909d9da263164ab4d39614e0c50a32774a49b3390a53ffa63e8d74b8e7c0b
Packaging & OEM	h8e30701845bea3e44d0aed1ba6d4893a0de91fea6f42571d3714a3c6daa39978
Deployment	hd995f5ddfb1625e3a33b0ee123b6672f35df88d6652eaec51d26f3a50b030ad8
Recall	hdf0f326b1bf6611d944491d7a0618af56ac57e391ba38425f9f33cafdd7439a9

Table 5.2: `currOwnerID` for Different Lifecycles.

Algorithm 1 Lifecycle Transition Algorithm

```
Input: Transition ID (TID), Transition Request $\mathcal{R}$
Output: Result (true or false)                                  ▷ Not visible externally
 1: procedure TransitionLifecycle(TID, $\mathcal{R}$)
 2:     if lifecycleMem.next[TID.value] == TID and $\mathcal{R}$ then
 3:         lifecycteSTS ← lifecycleMem.next[TID.lc]            ▷ Transition to lifecycle
 4:         return true
 5:     else
 6:         lifecycteSTS ← lifecycleMem.curr[currOwnerID.lc]
 7:         return false
 8:     end if
 9:     //Execute re-boot sequence in lifecycteSTS              ▷ Re-boot in current lifecycle
10: end procedure
```

Algorithm 2 Lifecycle Authentication Algorithm

Input: `currOwnerID` (`CID`), Boot Request $\mathcal{B}$
Output: Result (true or false)

```
 1: procedure LCAuthatBoot(CID, $\mathcal{B}$)
 2:     initialize MCSE                                        ▷ Power on and de-assert reset
 3:     if lcAuth.ID[value] == CID and $\mathcal{B}$ then
 4:         //Execute ChipID Authentication Routine                ▷ Authenticates ChipID
 5:         return true
 6:     else
 7:         lifecycteSTS $\leftarrow$ lifecycleMem.prev[currOwnerID.lc]   ▷ Boot in previous lifecycle
 8:         return false
 9:     end if
10:     AssetAccessPolicy(lifecycteSTS)             ▷ Enables differential access of assets
11: end procedure
```

5.4 Other Identity Authentication

...To be written...

Chapter 6

Firmware Signing and Authentication

6.1 Firmware Signing

- Vendor calculates the `XOR` of `IPAD` and Firmware Signing Key[1]:
  `IPAD_XOR = IPAD` $\oplus$ Firmware Signing Key
- Vendor calculates the `XOR` of `OPAD` and Firmware Signing Key:
  `OPAD_XOR = OPAD` $\oplus$ Firmware Signing Key
- The concatenation of `IPAD_XOR`, `OPAD_XOR`, and firmware binary is passed through `SHA256` module to generate 256-bits firmware signature:
  $\text{SHA-256}_{\text{input}} =$ `OPAD_XOR, Firmware Binary, IPAD_XOR`
  $\text{Firmware Signature} = \text{SHA-256}_{\text{output}}$
- The calculated 256-bit signature is then appended to the firmware binary to generate the firmware image:
  $\text{Firmware Image} = \{\text{Firmware Binary, Firmware Signature}\}$

At the end of the signing process, the firmware image payload ($\mathcal{I}$) of n bits for authentication transforms as follows:

$\mathcal{I} \to \mathcal{I}'$, where $\mathcal{I}' = \mathcal{I} + \mathcal{S}$, where $\mathcal{S}$ is the Firmware signature. $\mathcal{S}$ is the LSB [0:255], and the remainder of the binary is the firmware image.

6.2 Firmware Authentication

In the presence of a firmware upgrade request, TA2 will forward the request to `MCSE` for processing. Since this version of `MCSE` does not have an active connection with the AMI, firmware authentication requests have to be routed via the TA2 to `MCSE` for authentication.

The firmware authentication mechanism is built upon the HMAC algorithm and uses a 256-bit golden firmware signing key. The asset is transferred to `MCSE` during secure asset

[1] `IPAD` = `0x36` and `OPAD` = `0x5C` are predefined constants in HMAC.

provisioning[2] and stored in `MCSE's` secure memory. Below are the steps involved in verifying the firmware signature alongside its associated binary:

- TA2 initiates the process by signaling `MCSE` using `GPIO pin[14]`, indicating the presence of a firmware image that needs to be authenticated.
- `MCSE` executes the necessary operations to validate the authenticity of the firmware (see Algorithm 3).
- Upon completion of the authentication process, `MCSE` communicates the outcome to TA2 using `GPIO pin[15]` for successful authentication and `GPIO pin[16]` for a failed authentication.

Algorithm 3 Firmware Authentication Algorithm

```
Input: Firmware Authentication Request ($\mathcal{R}$), Firmware Image[N+M-1:0] ($\mathcal{I}$)
// Firmware Image = {Firmware Binary[N-1:0], Firmware Signature[M-1:0]}
Output: Authentication Success or Failure
 1: procedure FirmwareAuthentication($\mathcal{R}$, $\mathcal{I}$)
 2:     if $\mathcal{R}$ then
 3:         // Begin firmware image extraction
 4:         ChallengeFirmwareSignature ← $\mathcal{I}[M-1:0]$
 5:         FirmwareBinary ← $\mathcal{I}[N-1:M]$
 6:         // Fetch FirmwareSigningKey
 7:         IPAD_XOR ← IPAD ⊕ FirmwareSigningKey
 8:         OPAD_XOR ← OPAD ⊕ FirmwareSigningKey
 9:         $\text{SHA-256}_{\text{input}}$ ← Concatenate{OPAD_XOR, FirmwareBinary, IPAD_XOR}
10:         GoldenFirmwareSignature ← $\text{SHA-256}_{\text{output}}$
11:         if ChallengeFirmwareSignature == GoldenFirmwareSignature then
12:             Return Authentication Success
13:         else
14:             Return Authentication Failure
15:         end if
16:     end if
17:     Poll for Firmware Authentication Request
18: end procedure
```

[2] This occurs during the Manufacture and Test lifecycle where the device is under control of the HSM.

Chapter 7

Scan Protection

...To be written...