**Overview**

This document specifies the functionality of the OpenTitan key manager.

**Features**

* One-way key and identity (working) state hidden from software.
* Version controlled identity and key generation.
* Key generation for both software consumption and hardware sideload.
* Support for DICE open profile.

**Description**

The key manager implements the hardware component of the [identities and root keys](https://docs.opentitan.org/doc/security/specs/identities_and_root_keys/) strategy of OpenTitan.

It enables the system to shield critical assets from software directly and provides a simple model for software to use derived key and identity outputs.

**Theory of Operation**

In the diagram, the red boxes represent the working state and the associated internal key, the black ovals represent derivation functions, the green squares represent software inputs, and the remaining green / purple shapes represent outputs to both software and hardware.

In OpenTitan, the derivation method selected is [KMAC](https://github.com/lowRISC/opentitan/blob/master/hw/ip/kmac/README.md). Each valid operation involves a KMAC invocation using the key manager internal key and other HW / SW supplied inputs as data. While KMAC can generate outputs of arbitrary length, this design fixes the size to 256b.

Effectively, the key manager behavior is divided into 3 classes of functions

* Key manager state advancement
  + The results are never visible to software and not directly usable by any software controlled hardware
* Output key generation
  + Results can be visible to software or consumed by hardware (sideload)
* Identity / seed generation
  + Results are always visible to software and used for asymmetric cryptography

In general, the key generation and seed generation functions are identical. They differ only in how software chooses to deploy the outputs.

For clarity, all commands issued to the key manager by software are referred to as operations. Transactions refer to the interaction between key manager and KMAC if a valid operation is issued.

**Key Manager State**

The key manager working state (red boxes in the functional model) represents both the current state of the key manager as well as its related internal key. Each valid state (Initialized / CreatorRootKey / OwnerIntermediateKey / OwnerRootKey), supplies its secret material as the "key" input to a KMAC operation. Invalid states, such as Reset / Disabled on the other hand, either do not honor operation requests, or supplies random data when invoked.

The data input is dependent on each state, see below.

**Reset**

To begin operation, the state must first transition to Initialize. The advancement from Reset to Initialized is irreversible during the current power cycle. Until the initialize command is invoked, the key manager rejects all other software commands.

**Initialized**

When transitioning from Reset to Initialized, random values obtained from the entropy source are used to populate the internal key first. Then the root key stored in OTP, if valid, is loaded into the internal key. This ensures that the hamming delta from the previous value to the next value is non-deterministic. The advancement from Initialized to CreatorRootKey is irreversible during the current power cycle.

**CreatorRootKey**

CreatorRootKey is the first operational state of the key manager. When transitioning from Initialized to this state, a KMAC operation is invoked using the RootKey as the key (from OTP), and the remaining inputs as data. The output of the KMAC operation replaces the previous value of the internal key, and the new value becomes the CreatorRootKey.

Inputs to the derivation function are:

* HardwareRevisionSecret: A global design time constant.
* ROMHash: SHA-3-256 hash of the ROM image.
* HealthMeasurement: Current life cycle state
  + To avoid a state value corresponding to each life cycle state, the raw life cycle value is not used.
  + Instead, certain life cycle states diversify the same way.
  + Please see the life cycle controller for more details.
* DeviceIdentifier: Unique device identification.
* SoftwareBinding: A software programmed value related to ROMExt.

Other than HardwareRevisionSecret, none of the values above are considered secret.

Once the CreatorRootKey is reached, software can request key manager to advance state, generate output key or generate output identity. The key used for all 3 functions is the CreatorRootKey.

The advancement from CreatorRootKey to the OwnerIntermediateKey is irreversible during the current power cycle.

Keymgr reads the root key from OTP in a single clock cycle. It assumes that when keymgr's internal FSM reaches to this clock cycle, OTP root key is already available (valid is set to 1). Otherwise, keymgr skips loading the root key.

**OwnerIntermediateKey**

This is the second operational state of the key manager. This state is reached through another invocation of the KMAC operation using the previous internal key, and other inputs as data. The output of the KMAC operation replaces the previous value of the internal key, and the new value becomes the OwnerIntermediateKey.

The relevant data inputs are:

* CreatorSecret: A secret seed from flash determined by the SiliconCreator.
* SoftwareBinding: A software programmed value representing the first owner code to be run.

Once the OwnerIntermediateKey is created, software can request key manager to advance state, generate output key or generate output identity. The key used for all 3 functions is the OwnerIntermediateKey.

The advancement from OwnerIntermediateKey to the OwnerRootKey is irreversible during the current power cycle.

**OwnerRootKey**

This is the last operational state of the key manager. This state is reached through another invocation of the KMAC operation using the previous internal key, and other inputs as data. The output of the KMAC operation replaces the previous value of the internal key, and the new value becomes the OwnerRootKey.

The relevant inputs are:

* OwnerRootSecret: Secret seed from flash.
* SoftwareBinding - A software programmed value representing the owner kernel code.

Once the OwnerRootKey is created, software can request key manager to advance state, generate output key or generate output identity. An advance command invoked from OwnerRootKey state simply moves the state to Disabled.

The generate output and generate identity functions use OwnerRootKey as the KMAC key. The advancement from OwnerRootKey to the Disabled is irreversible during the current power cycle.

**Disabled**

Disabled is a state where the key manager is no longer operational. Upon Disabled entry, the internal key is updated with KMAC computed random values; however, previously generated sideload key slots and software key slots are preserved. This allows the software to keep the last valid keys while preventing the system from further advancing the valid key.

When advance and generate calls are invoked from this state, the outputs and keys are indiscriminately updated with randomly computed values. Key manager enters disabled state based on direct invocation by software:

* Advance from OwnerRootKey
* Disable operation

**Invalid**

Invalid state is entered whenever key manager is deactivated through the [life cycle connection](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#life-cycle-connection) or when an operation encounters a [fault](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#faults-and-operational-faults) . Upon Invalid entry, the internal key, the sideload key slots and the software keys are all wiped with entropy directly.

**Invalid Entry Wiping**

Since the life cycle controller can deactivate the key manager at any time, the key manager attempts to gracefully handle the wiping process. When deactivated, the key manager immediately begins wiping all keys (internal key, hardware sideload key, software key) with entropy. However, if an operation was already ongoing, the key manager waits for the operation to complete gracefully before transitioning to invalid state.

While waiting for the operation to complete, the key manager continuously wipes all keys with entropy.

**Invalid and Disabled State**

Invalid and Disabled states are functionally very similar. The main difference between the two is "how" the states were reached and the entry behavior.

Disabled state is reached through intentional software commands where the sideload key slots and software key are not wiped, while Invalid state is reached through life cycle deactivation or operational faults where the internal key, sideload key slots and software key are wiped.

This also means that only Invalid is a terminal state. If after entering Disabled life cycle is deactivated or a fault is encountered, the same [invalid entry procedure](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#Invalid) is followed to bring the system to a terminal Invalid state.

If ever multiple conditions collide (a fault is detected at the same time software issues disable command), the Invalid entry path always takes precedence.

**Life Cycle Connection**

The function of the key manager is directly managed by the [life cycle controller](https://github.com/lowRISC/opentitan/blob/master/hw/ip/lc_ctrl/README.md#key_manager_en).

Until the life cycle controller activates the key manager, the key manager does not accept any software commands. Once the key manager is activated by the life cycle controller, it is then allowed to transition to the various states previously [described](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#key-manager-states).

When the life cycle controller deactivates the key manager, the key manager transitions to the Invalid state.

**Commands in Each State**

During each state, there are 3 valid commands software can issue:

* Advance state
* Output generation
* Identity generation

The software is able to select a command and trigger the key manager FSM to process one of the commands. If a command is valid during the current working state, it is processed and acknowledged when complete.

If a command is invalid, the behavior depends on the current state. If the current state is Reset, the invalid command is immediately rejected as the key manager FSM has not yet been initialized. If the current state is any other state, the key manager sequences random, dummy data to the KMAC module, but does not update internal key, sideload key slots or software keys. For each valid command, a set of inputs are selected and sequenced to the KMAC module.

During Disable and Invalid states, the internal key, sideload key slots and software key are updated based on the input commands as with normal states. There are however a few differences:

* The updates are made regardless of any error status to ensure their values are further scrambled.
* Instead of normal input data, random data is selected for KMAC processing.
* All operations return an invalid operations error, in addition to any other error that might naturally occur.

**Generating Output Key**

The generate output command is composed of 2 options

* Generate output key for software, referred to as generate-output-sw
* Generate output key for hardware, referred to as generate-output-hw

The hardware option is meant specifically for symmetric sideload use cases. When this option is issued, the output of the KMAC invocation is not stored in software visible registers, but instead in hardware registers that directly output to symmetric primitives such as AES, KMAC and OTBN.

**KMAC Operations**

All invoked KMAC operations expect the key in two shares. This means the internal key, even though functionally 256b, is maintained as 512b. The KMAC processed outputs are also in 2-shares. For generate-output-sw commands, software is responsible for determining whether the key manager output should be preserved in shares or combined.

**Errors, Faults and Alerts**

The key manager has two overall categories of errors:

* Recoverable errors
* Fatal errors

Recoverable errors are those likely to have been introduced by software and not fatal to the key manager or the system. Fatal errors are logically impossible errors that have a high likelihood of being a fault and thus fatal.

Each category of error can be further divided into two:

* Synchronous errors
* Asynchronous errors

Synchronous errors happen only during a key manager operation. Asynchronous errors can happen at any time.

Given the above, we have 4 total categories of errors:

* Synchronous recoverable errors
* Asynchronous recoverable errors
* Synchronous fatal errors
* Asynchronous fatal errors

All recoverable errors (synchronous and asynchronous) are captured in [ERR\_CODE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code). All fatal errors (synchronous and asynchronous) are captured in [FAULT\_STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#fault_status).

Recoverable errors cause a recoverable alert to be sent from the key manager. Fatal errors cause a fatal alert to be sent from the key manager.

Below, the behavior of each category and its constituent errors are described in detail.

**Synchronous Recoverable Errors**

These errors can only happen when a key manager operation is invoked and are typically associated with incorrect software programming. At the end of the operation, key manager reports whether there was an error in [ERR\_CODE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code) and sends a recoverable alert.

* [ERR\_CODE.INVALID\_OP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code) Software issued an invalid operation given the current key manager state.
* [ERR\_CODE.INVALID\_KMAC\_INPUT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code) Software supplied invalid input (for example a key greater than the max version) for a key manager operation.

**Asynchronous Recoverable Errors**

These errors can happen at any time regardless of whether there is a key manager operation. The error is reported in [ERR\_CODE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code) and the key manager sends a recoverable alert.

* [ERR\_CODE.INVALID\_SHADOW\_UPDATE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code) Software performed an invalid sequence while trying to update a key manager shadow register.

**Synchronous Fatal Errors**

These errors can only happen when a key manager operation is invoked and receives malformed operation results that are not logically possible. At the end of the operation, key manager reports whether there was an error in [FAULT\_STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#fault_status) and continuously sends fatal alerts .

Note, these errors are synchronous from the perspective of the key manager, but they may be asynchronous from the perspective of another module.

**Asynchronous Fatal Errors**

These errors can happen at any time regardless of whether there is a key manager operation. The error is reported in [FAULT\_STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#fault_status) and the key manager continuously sends fatal alerts.

**Faults and Operational Faults**

When a fatal error is encountered, the key manager transitions to the Invalid [state](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#invalid-entry-wiping). The following are a few examples of when the error occurs and how the key manager behaves.

**Example 1: Fault During Initialization**

The key manager is in the Reset state and receives an advance operation. After it has reseeded its internal PRNG with entropy, it will try to load the Creator Root Key from OTP. If not both shares of the Creator Root Key are valid at that point, key manager will wipe its secrets and transition to the Invalid state.

**Example 2: Fault During Operation**

The key manager is running a generate operation and a non-onehot command was observed by the KMAC interface. Since the non-onehot condition is a fault, it is reflected in [FAULT\_STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#fault_status) and a fatal alert is generated. The key manager transitions to Invalid state, wipes internal storage and reports an invalid operation in [ERR\_CODE.INVALID\_OP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code).

**Example 3: Fault During Idle**

The key manager is NOT running an operation and is idle. During this time, a fault is observed on the regfile (shadow storage error) and FSM (control FSM integrity error). The faults are reflected in [FAULT\_STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#fault_status). The key manager transitions to Invalid state, wipes internal storage but does not report an invalid operation.

**Example 4: Operation after Fault Detection**

Continuing from the example above, the key manager now begins an operation. Since the key manager is already in Invalid state, it does not wipe internal storage and reports an invalid operation in [ERR\_CODE.INVALID\_OP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#err_code).

**Additional Details on Invalid Input**

What is considered invalid input changes based on current state and operation.

When an advance operation is invoked:

* The internal key is checked for all 0's and all 1's.
* During the Reset state, both shares of the the Creator Root Key provided by OTP are checked to be valid.
* During Initialized state, creator seed, device ID and health state data is checked for all 0's and all 1's.
* During CreatorRootKey state, the owner seed is checked for all 0's and all 1's.
* During all other states, nothing is explicitly checked.

When a generate output key operation is invoked:

* The internal key is checked for all 0's and all 1's.
* The key version is less than or equal to the max key version.

When a generate output identity is invoked:

* The internal key is checked for all 0's and all 1's.

**Invalid Operation**

The table below enumerates the legal operations in a given state. When an illegal operation is supplied, the error code is updated and the operation is flagged as done with error.

| **Current State** | **Legal Operations** |
| --- | --- |
| Reset | Advance |
| Initialized | Disable / Advance |
| CreatorRootKey | Disable / Advance / Generate |
| OwnerIntKey | Disable / Advance / Generate |
| OwnerRootKey | Disable / Advance / Generate |
| Invalid/Disabled | None |

* All operations invoked during Invalid and Disabled states lead to invalid operation error.

**Error Response**

In addition to alerts and interrupts, key manager may also update the internal key and relevant outputs based on current state. See the tables below for an enumeration.

| **Current State** | **Invalid States** | **Invalid Output** | **Invalid Input** | **Invalid Operation** |
| --- | --- | --- | --- | --- |
| Reset | Not Possible | Not Possible | Not possible | Not updated |
| Initialized | Updated | Updated | Not updated | Not updated |
| CreatorRootKey | Updated | Updated | Not updated | Not possible |
| OwnerIntKey | Updated | Updated | Not updated | Not possible |
| OwnerRootKey | Updated | Updated | Not updated | Not possible |
| Invalid/Disabled | Updated | Updated | Updated | Updated |

* During Reset state, the KMAC module is never invoked, thus certain errors are not possible.
* During Initialized, CreatorRootKey, OwnerIntermediateKey and OwnerRootKey states, a fault error causes the relevant key / outputs to be updated; however an operational error does not.
* During Invalid and Disabled states, the relevant key / outputs are updated regardless of the error.
* Only the relevant collateral is updated -> ie, advance / disable command leads to working key update, and generate command leads to software or sideload key update.
* During Disabled state, if life cycle deactivation or an operational fault is encountered, the key manager transitions to Invalid state, see [here](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/theory_of_operation.md#invalid-and-disabled-state)

**DICE Support**

The key manager supports [DICE open profile](https://pigweed.googlesource.com/open-dice/+/HEAD/docs/specification.md#Open-Profile-for-DICE). Specifically, the open profile has two compound device identifiers.

* Attestation CDI
* Sealing CDI

The attestation CDI is used to attest hardware and software configuration and is thus expected to change between updates. The sealing CDI on the other hand, is used to attest the authority of the hardware and software configuration. The sealing version is thus expected to remain stable across software updates.

To support these features, the key manager maintains two versions of the working state and associated internal key. There is one version for attestation and one version for sealing.

The main difference between the two CDIs is the different usage of SW\_BINDING. For the Sealing CDI, the ["SEALING\_SW\_BINDING"](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#sealing_sw_binding) is used, all other inputs are the same. For the Attestation CDI, the ["ATTEST\_SW\_BINDING"](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#attest_sw_binding) is used, all other inputs are the same.

When invoking an advance operation, both versions are advanced, one after the other. There are thus two KMAC transactions. The first transaction uses the Sealing CDI internal key, ["SEALING\_SW\_BINDING"](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#sealing_sw_binding) and other common inputs. The second transaction uses the Attestation CDI internal key, ["ATTEST\_SW\_BINDING"](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#attest_sw_binding) and other common inputs.

When invoking a generate operation, the software must specify which CDI to use as the source key. This is done through ["CONTROL.CDI\_SEL"](https://github.com/lowRISC/opentitan/blob/master/hw/ip/keymgr/doc/registers.md#control). Unlike the advance operation, there is only 1 KMAC transaction since we pick a specific CDI to operate.

When disabling, both versions are disabled together.