# Integration of a Cryptographic API with Configurable Hardware and Software Backends in RIOT

Lena Boeckmann

September 8, 2021

# Crypto Support in RIOT

- RIOT provides built-in software implementations and third party libraries
- Only pure software implementations
- Run on supported platforms, but are quite inefficient

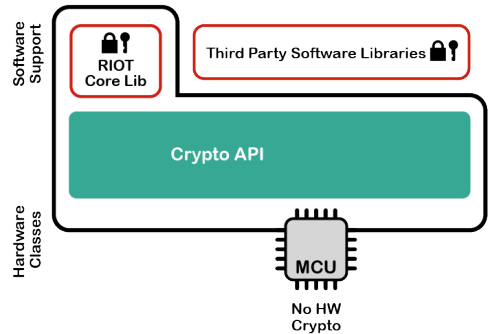
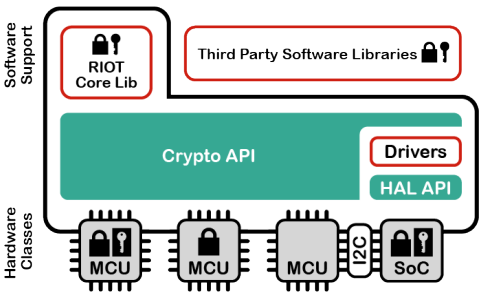



Figure: Currently supported crypto backends in RIOT

# Cryptographic Backends

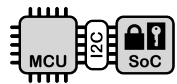




Crypto performed by software libraries

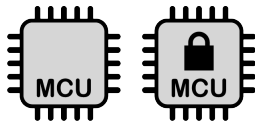

Crypto performed by hardware without key storage


Crypto performed by hardware with key storage




Crypto performed by external hardware with key storage

# Transparent Drivers for Unprotected Keys

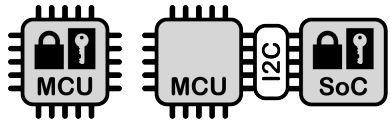



- Keys are stored in RAM or ROM
- Can be operated on by any implementation that accepts plain text key material
- User can freely choose between hardware or software implementations

**Definition**

**Transparent drivers** are software implementations and drivers that can be invoked without being dependent on the actual location of a key.

# Opaque Drivers for Protected Keys



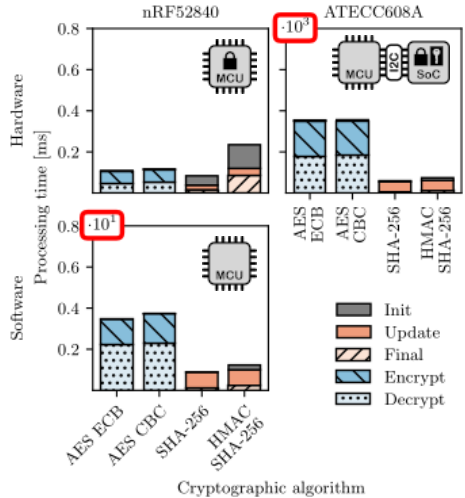

- Dedicated key storage areas (often slots)
- Only crypto processor can access key material
- Caller provides key identifier or slot number
- Only driver assigned to processor the key is located in can be invoked

**Definition**

**Opaque drivers** operate on protected keys. They are bound to a key location and can not be chosen freely.

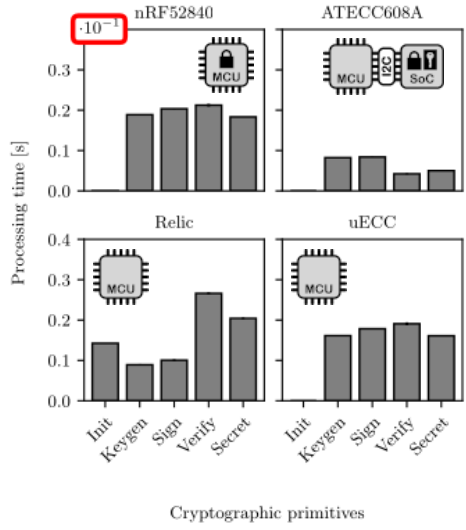
# Benefits of Hardware Crypto in Symmetric Operations

- Accelerator on nRF52840 is faster than RIOT software implementation
- Secure Element ATECC608A is less efficient, comes with benefit of protected storage



# Benefits of Hardware Crypto in Asymmetric Operations

- Secure Element ATECC608A is faster than software
- Accelerator on nRF52840 outperforms software and secure element



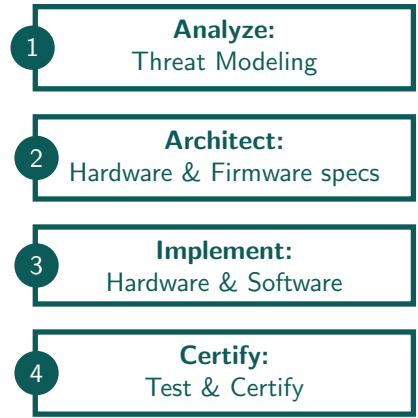
# Challenges and Solution

- Platforms with varying hardware crypto capabilities need to be supported
- Both protected and unprotected key storage need to be supported
- Hardware and software backends should be exchanged transparently beneath a unified API

**Our Solution:**
Implementing the ARM Platform Security Architecture Crypto API

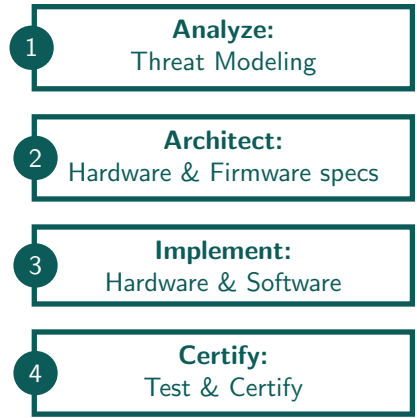
# What is PSA?

- ARM Platform Security Architecture



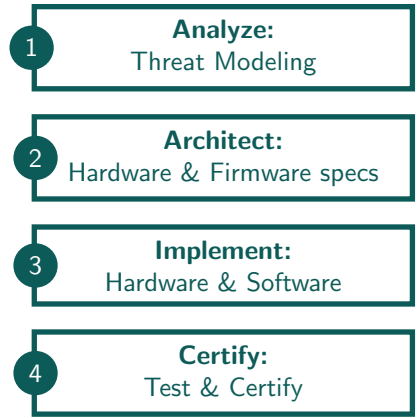
# What is PSA?

- ARM Platform Security Architecture
- Framework for development of secure IoT systems



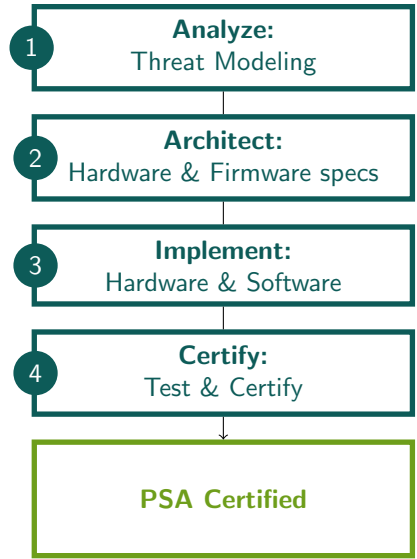
# What is PSA?

- ARM Platform Security Architecture
- Framework for development of secure IoT systems
- Threat models, specifications and reference implementations



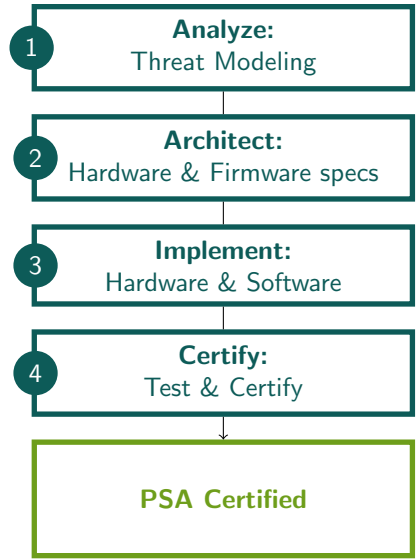
# What is PSA?

- ARM Platform Security Architecture
- Framework for development of secure IoT systems
- Threat models, specifications and reference implementations
- Implementations may be certified through PSA Certified scheme



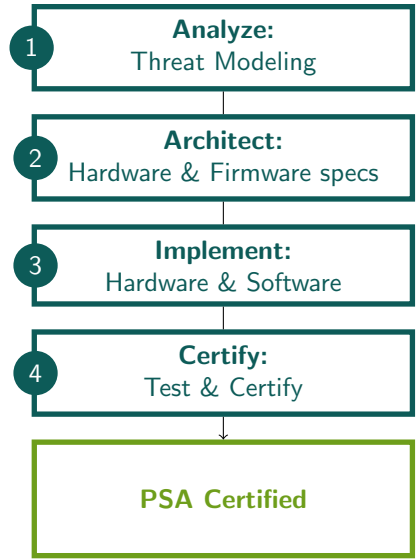
# What is PSA?

- ARM Platform Security Architecture
- Framework for development of secure IoT systems
- Threat models, specifications and reference implementations
- Implementations may be certified through PSA Certified scheme
- Open source test suite for implementation verification



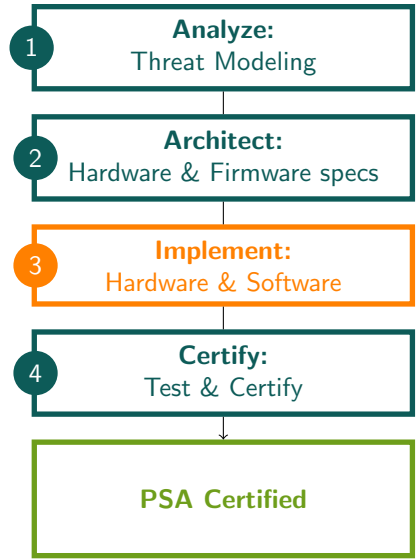
# What is PSA?

- Where can we find PSA Crypto?

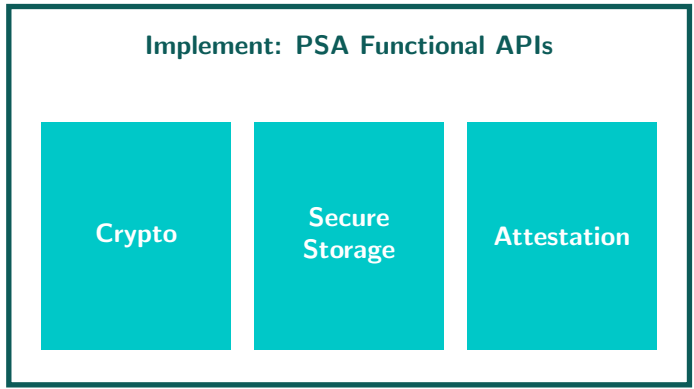


# What is PSA?

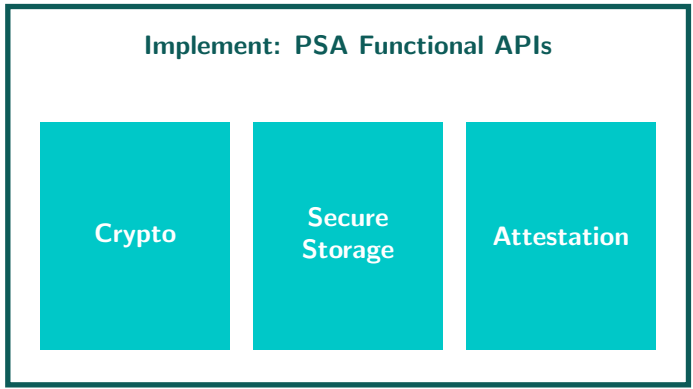
- Where can we find PSA Crypto?



# What is the PSA Crypto API?

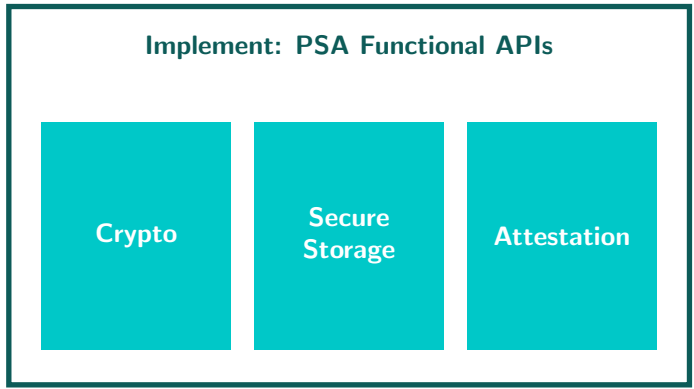


# What is the PSA Crypto API?




- Platform independent, suitable for IoT devices

# What is the PSA Crypto API?

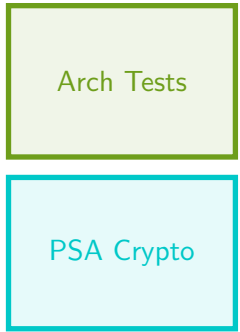



- Platform independent, suitable for IoT devices
- Designed with usability and portability in mind

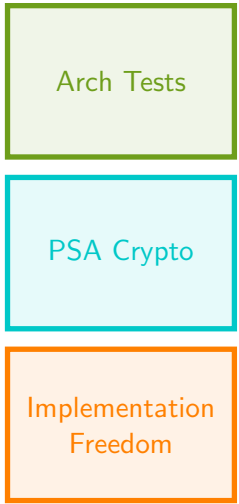
# What do we get?


Complete API design specification

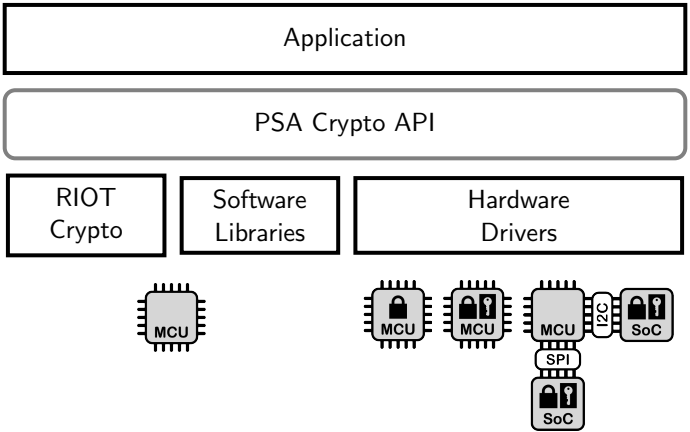
# What do we get?



# What do we get?



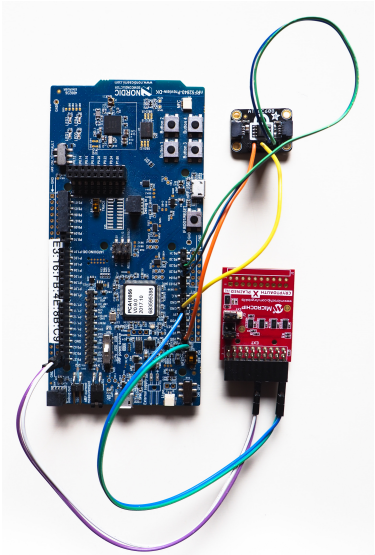
# Backend Flexibility




- Hardware agnostic application development possible
- Backends can be exchanged and combined as needed
- Use of multiple secure elements possible

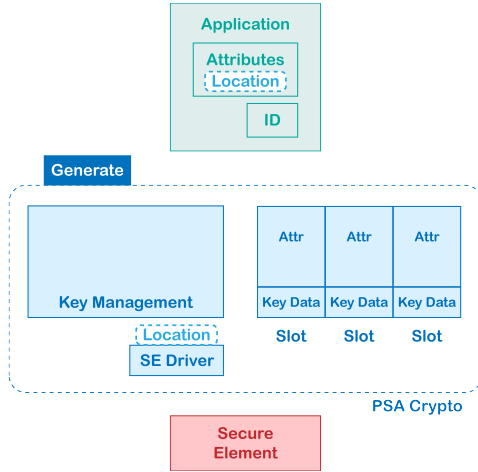
# Secure Element Handling

- Multiple SE's can be managed by a driver registry
- Calls are dispatched to appropiate driver depending on key location
- PSA Crypto reference implementation provides an API for SEs



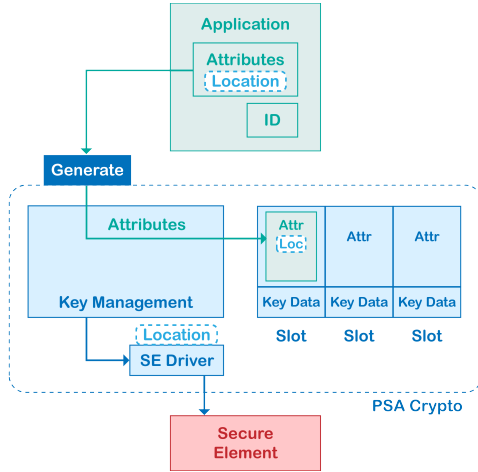
# Handling Keys in Protected Storage

1. Application provides key attributes with key location



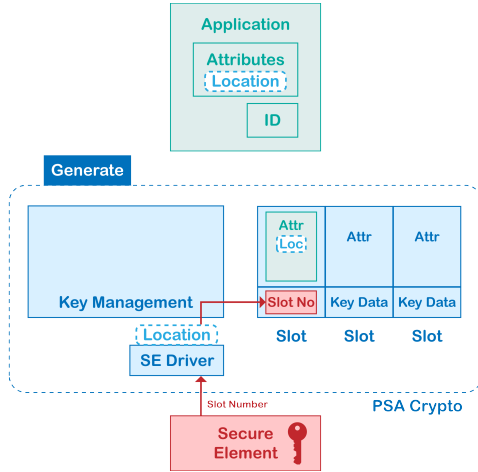
# Handling Keys in Protected Storage

1. Application provides key attributes with key location
2. Attributes are stored and opaque SE driver is invoked



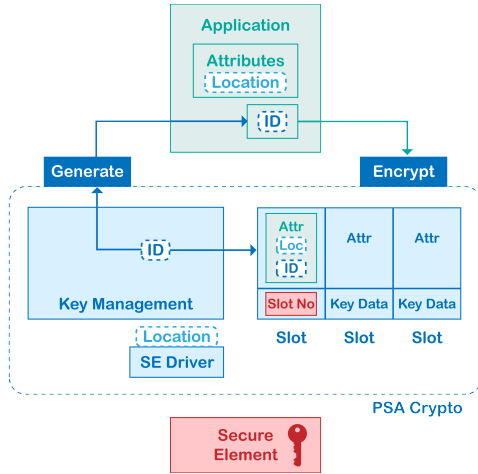
# Handling Keys in Protected Storage

1. Application provides key attributes with key location
2. Attributes are stored and opaque SE driver is invoked
3. SE generates and stores key in free slot, returns slot number



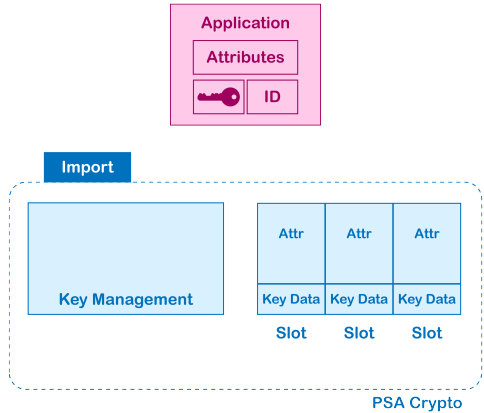
# Handling Keys in Protected Storage

1. Application provides key attributes with key location
2. Attributes are stored and opaque SE driver is invoked
3. SE generates and stores key in free slot, returns slot number
4. Key manager returns key ID to application for later use



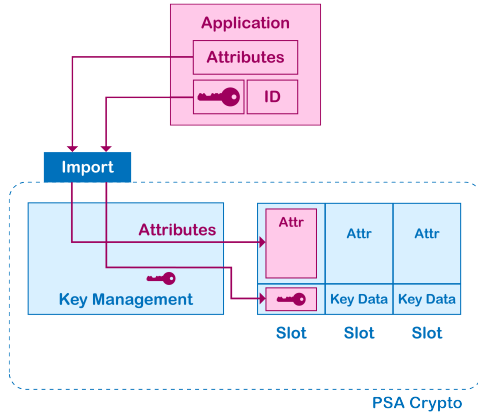
# Handling Externally Provided Keys

1. Application provides attributes and key material



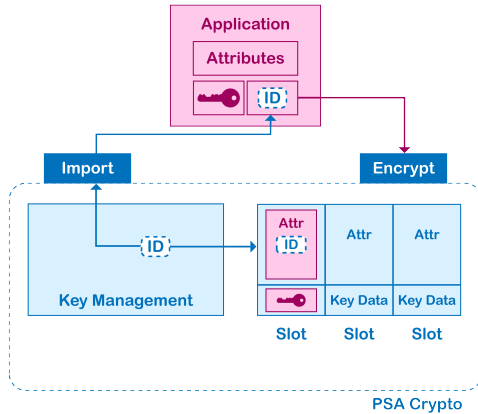
# Handling Externally Provided Keys

1. Application provides attributes and key material
2. Both are stored in local memory



# Handling Externally Provided Keys

1. Application provides attributes and key material
2. Both are stored in local memory
3. Key manager returns key ID for later use



# What is mbedTLS doing?

- Working on reference implementations for PSA Functional APIs
- We're following their progress and have been in contact
- Still under construction

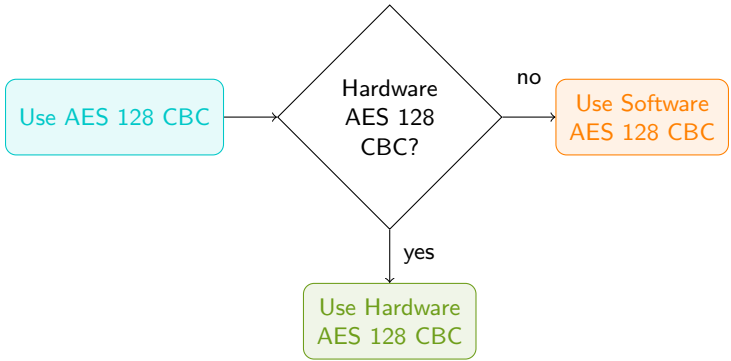
# Driver Wrapper Generation

**Support of different backends in mbedTLS**

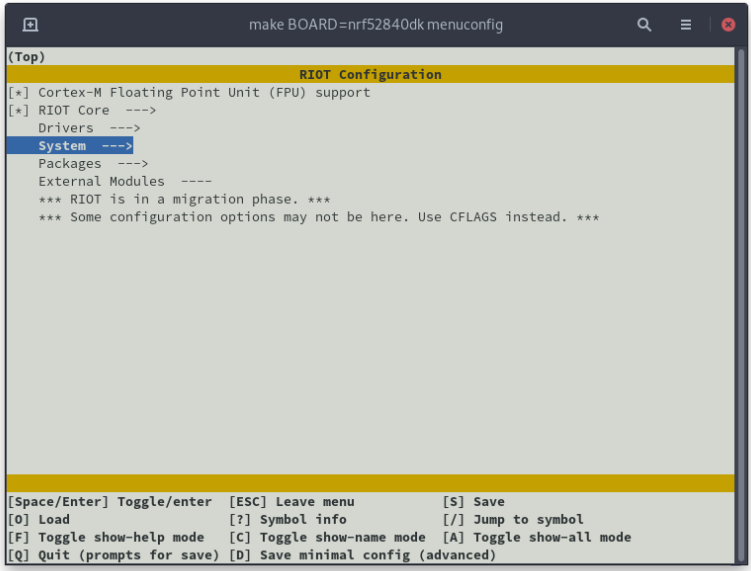
- Provide information about available drivers at compile time
- Driver description files in JSON format
- Code generator will parse descriptions and generate wrapper
- Wrapper contains calls to available drivers and software fallback
- Code generator does not exist, yet

# Configuring Crypto Backends with Kconfig

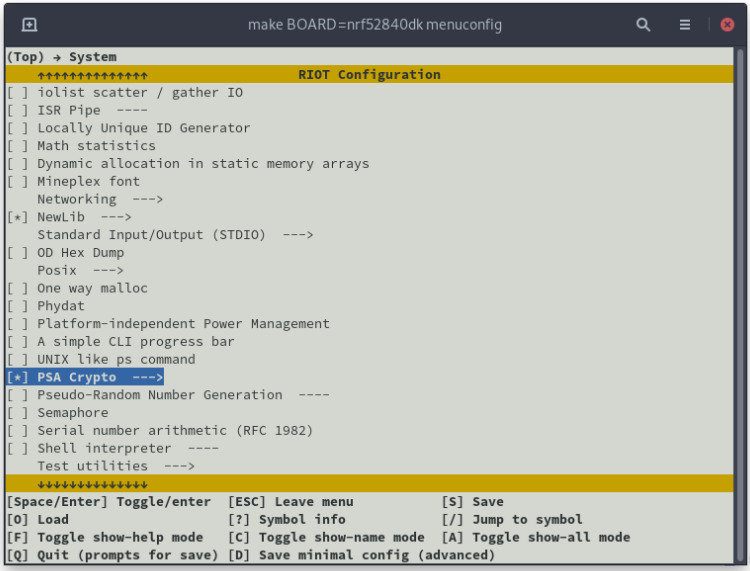
- If CPU defines symbol for hardware crypto (e.g. HAS_HW_AES_128_CBC), hardware is default backend
- User may change default configuration if desired



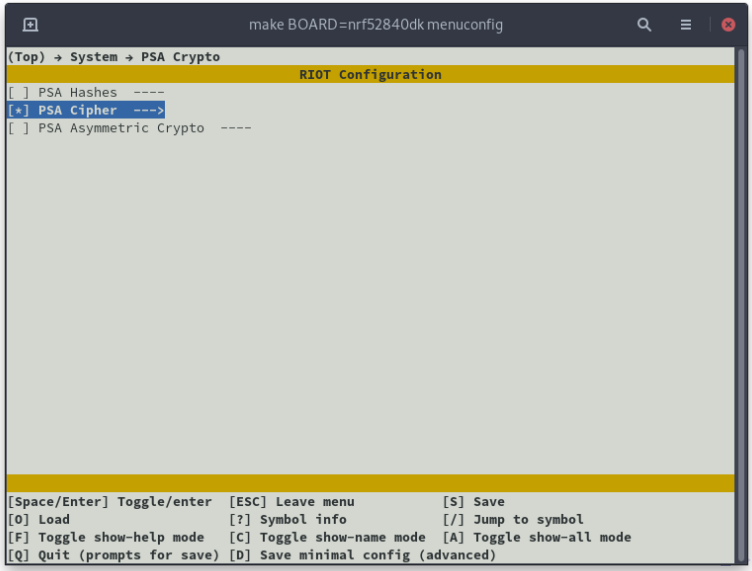
# Menuconfig Walkthrough



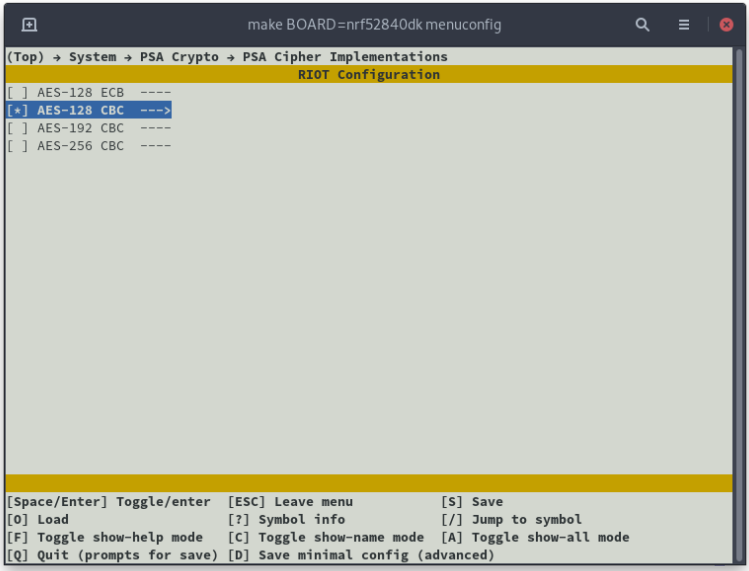
# Menuconfig Walkthrough



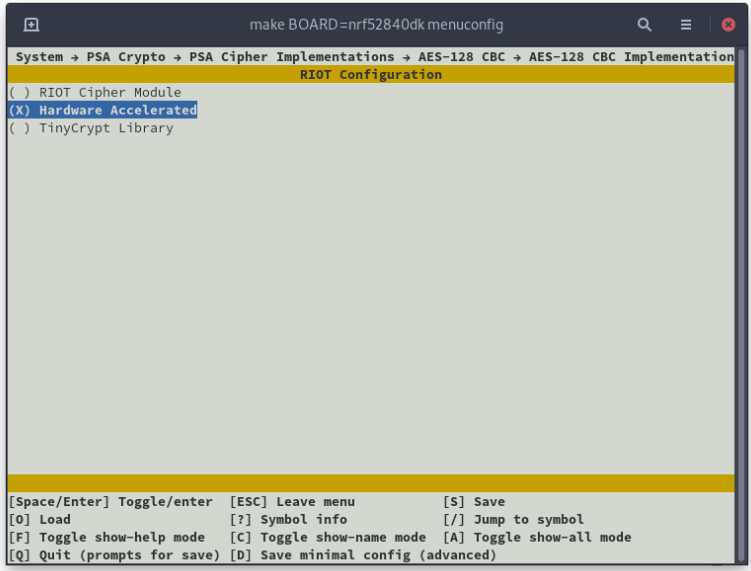
# Menuconfig Walkthrough



# Menuconfig Walkthrough

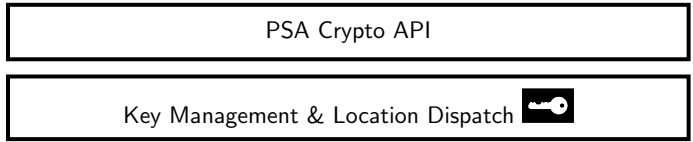


# Menuconfig Walkthrough

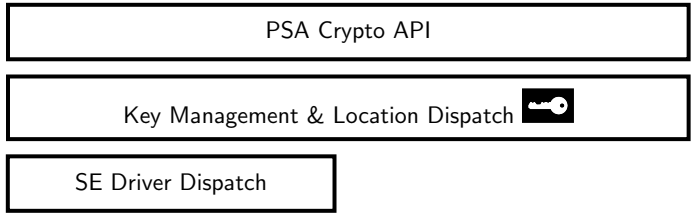


# PSA Crypto Implementation Structure

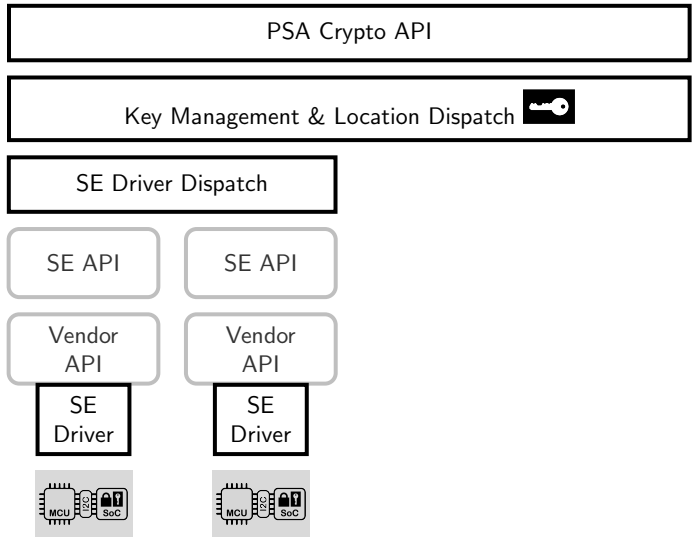
# PSA Crypto Implementation Structure



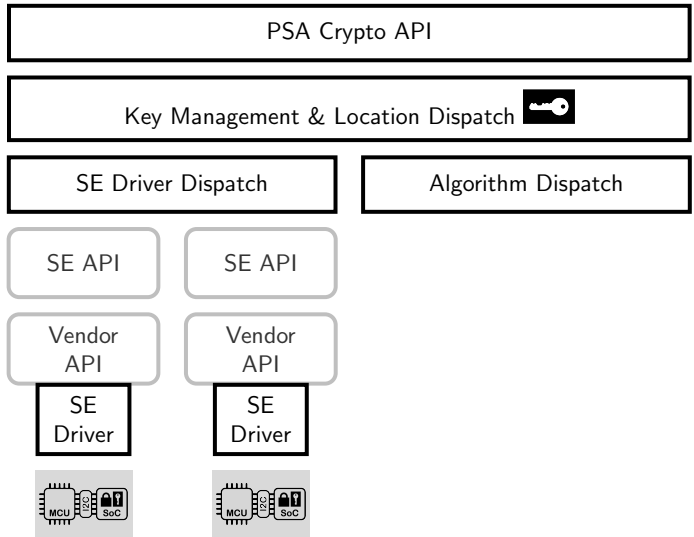
# PSA Crypto Implementation Structure



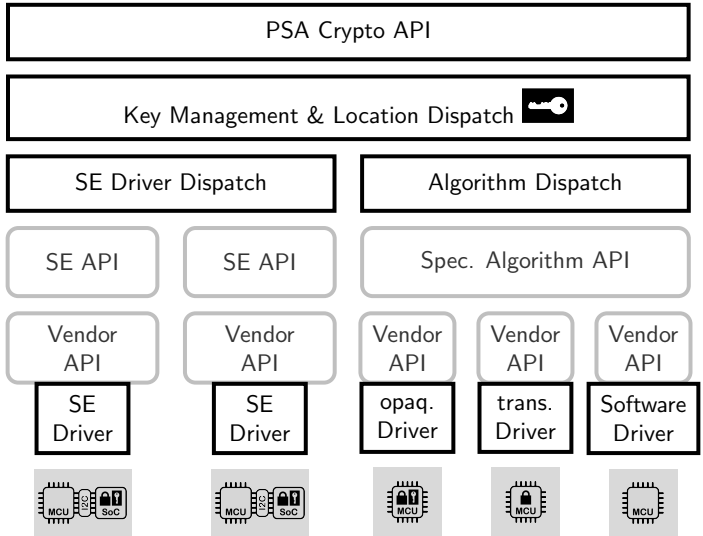
# PSA Crypto Implementation Structure



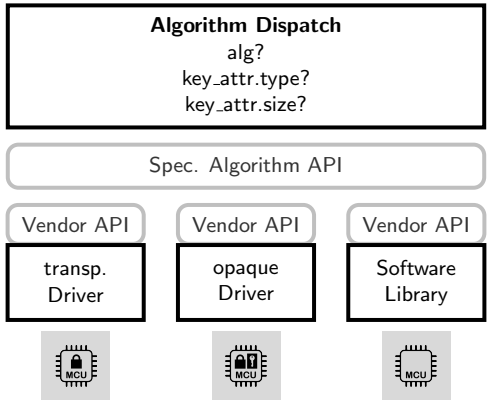
# PSA Crypto Implementation Structure



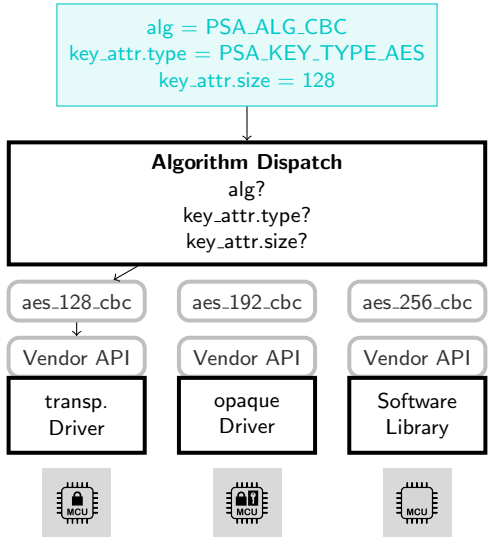
# PSA Crypto Implementation Structure



# Algorithm Dispatch



# Algorithm Dispatch



# Application Example of AES Encryption

```
uint8_t key[] = { ... };
uint8_t key_length = AES_128_KEY_LENGTH;

psa_key_id_t key_id;
psa_key_attributes_t attr = psa_key_attributes_init();

// Set key attributes
psa_key_lifetime_t lifetime = PSA_KEY_LOCATION_LOCAL_STORAGE |
                              PSA_KEY_PERSISTENCE_VOLATILE;

psa_set_key_lifetime(&attr, lifetime);
psa_set_key_algorithm(&attr, PSA_ALG_CBC_NO_PADDING);
psa_set_key_usage_flags(&attr, PSA_KEY_USAGE_ENCRYPT);
psa_set_key_type(&attr, PSA_KEY_TYPE_AES);
psa_set_key_bits(&attr, 128);

// Import key and store it as specified in lifetime
psa_import_key(&attr, key, key_length, &key_id);

// Encrypt some plaintext with key beloning to key_id
psa_cipher_encrypt(key_id, PSA_ALG_CBC_NO_PADDING, plaintext,
                   plaintext_length, output, output_size, &output_length);
```

# Implementation Status

| Component | Status | Description | Next Steps |
|---|---|---|---|
| PSA Arch Testsuite as package | ✔ | Hash tests work | Enable other algorithm tests |
| Volatile Key Management | ✔ | Volatile keys in local memory and SEs can be handled | Add support for persistent keys |
| Cryptographic Operations | ✔ | Some hashes and ciphers work | Extend support for cryptographic operations |
| Secure Element Handling | ✔ | Multiple devices can be handled | Add support for other devices |

**Outlook:** Support of operations in trusted execution environments