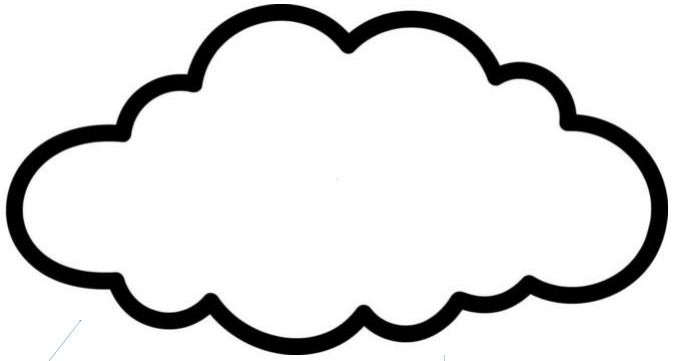
Remote Attestation

On how to improve trust between devices

Scenario

When would we want to use remote attestation?

Scenario

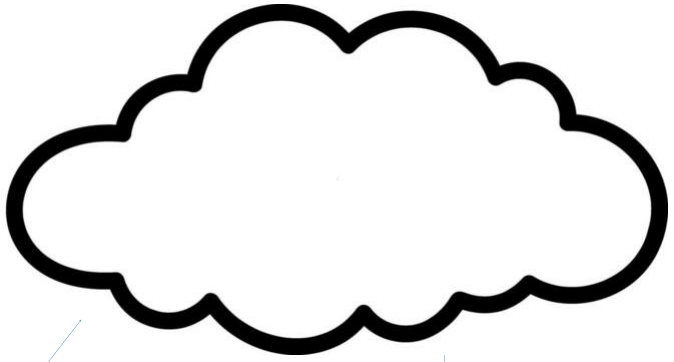


We used XYZ kWh for this period!

Here comes the client data!

Based on what the electric meter at the clients home reported, we file our bill.

Scenario



We used XYZ kWh for this period!

Here comes the client data!

Based on what the electric meter **at the clients home** reported, we file our bill.

Scenario


SURELY THE CUSTOMER HAS NO INCENTIVE TO TAMPER WITH THE METER.
YEAH RIGHT
RIGHT?

Solutions?

Solutions

- Just make the device hard to tamper with / physical protection!

- The attacker has time. Most if not all physical hindrances can be overcome with just enough time and effort

Solutions

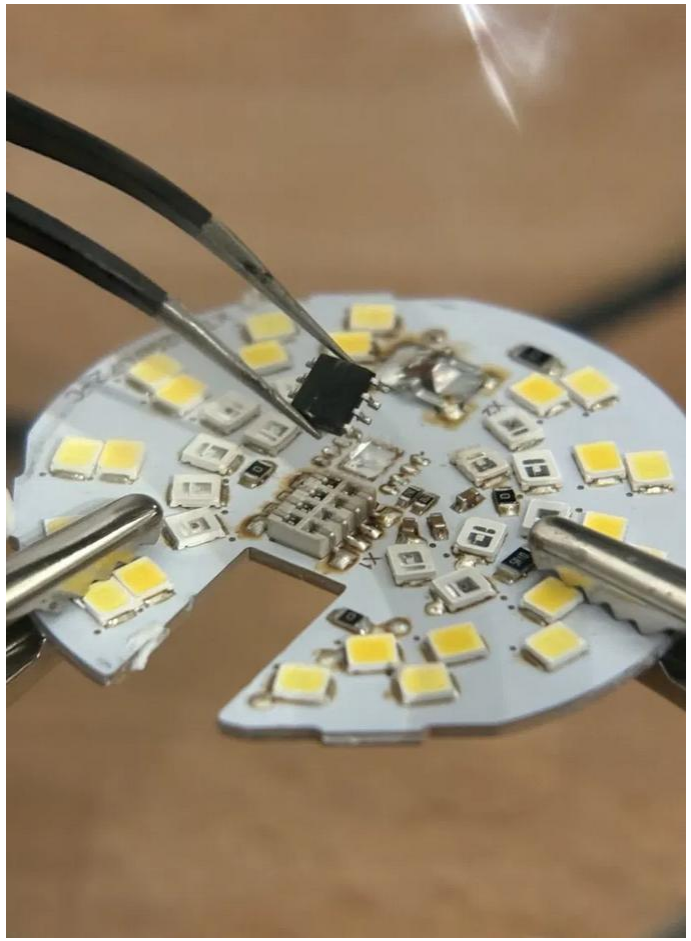
- Can‘t we use TLS to have encrypted communication?

- Yes, the communication might be encrypted but we still don‘t control what is being transmitted. The electric meter could still be lying

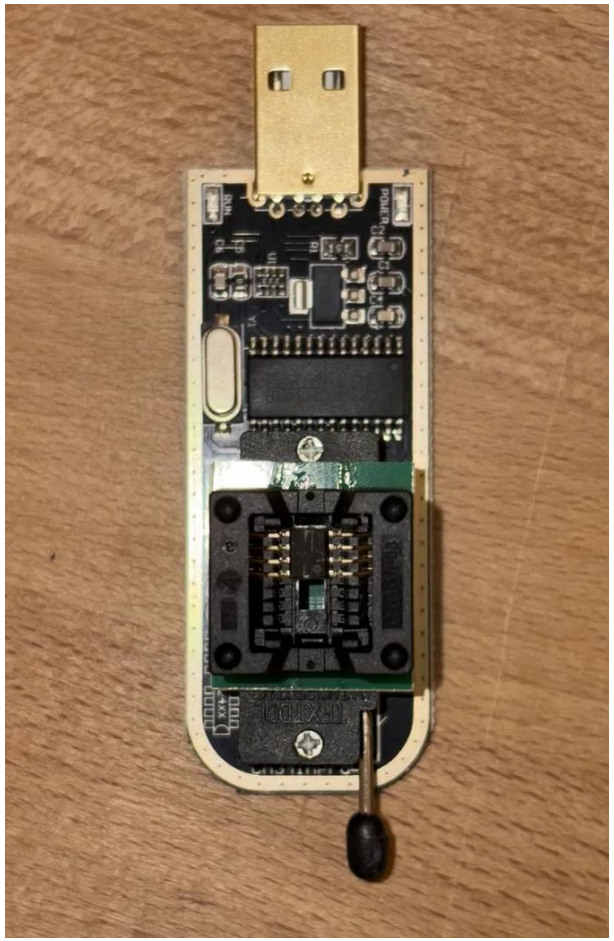
Solutions

- Okay, but why don‘t we just sign the measurements with a private key?

Solutions



40€ hot air soldering station



5€ CH341A SOP16 programmer

Solutions

- Hmm, my Windows does Secure Boot? Why don‘t we use that?

- Now the device can be assumed „secure“ but how do we expose that knowledge to the outside world?

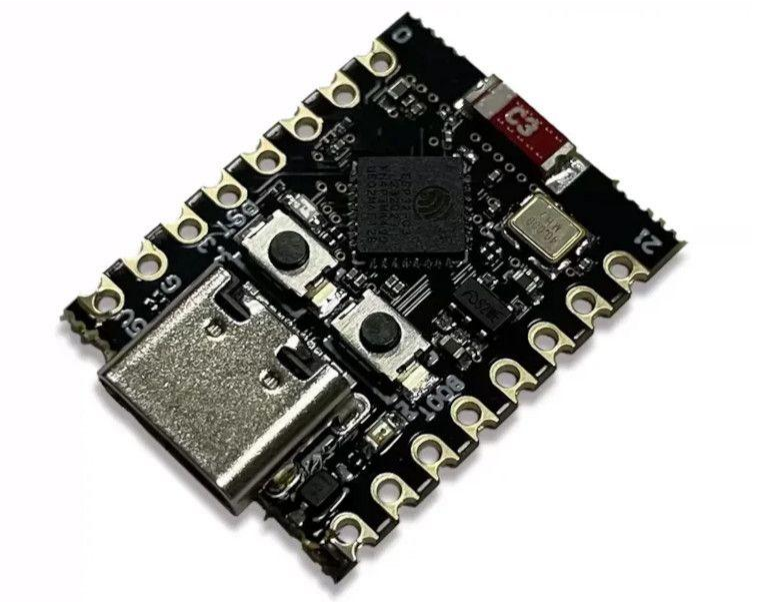
Solutions

- Okay okay, but what if we used secure boot and then put TLS on top of that?

- This doesn‘t work because secure boot saves the integrity of the boot image, not the confidentiality of the keys. Keys can still be extracted

Solutions

- I‘ve heard that chips like the esp32c3 support flash encryption?



- This makes extracting keys extremely hard. A bug in the network stack might still expose keys and compromise the entire system

A secure home for our keys

Both in Hardware and in Software

A secure home for our keys

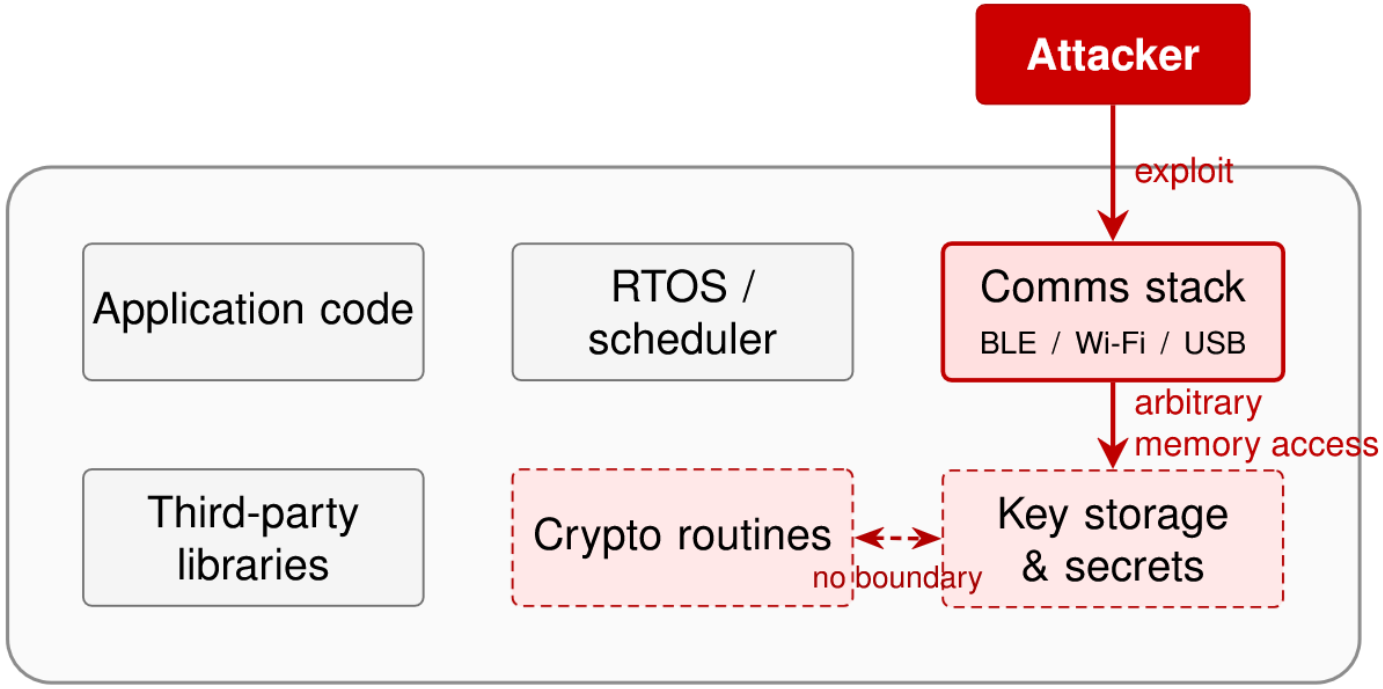
- We want a place that:
  - Is secured against physical tampering / flash extraction
  - Software bugs / flash dumps from the inside

A secure home for our keys

- In app variable -> same address space
- Separate Chip (TPM / SE) -> bigger BOM and potentially no bus encryption
- KMU -> the key never enters the address space. But it signs whatever it is handed
- MPU -> protects the OS from the app!

A secure home for our keys

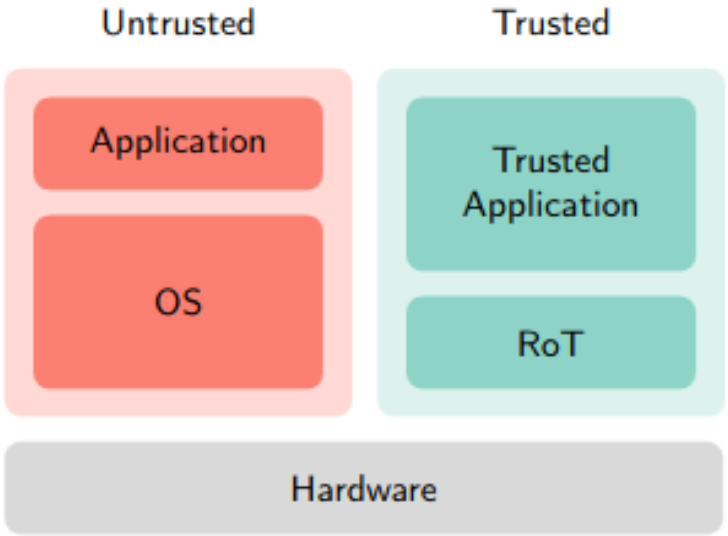
One shared world:
One firmware image, one privilege domain. A bug in any component (e.g. the network stack) lets an attacker read the keys and call the crypto routines directly



A secure home for our keys – TrustZone-M

Every address is either secure or non-secure

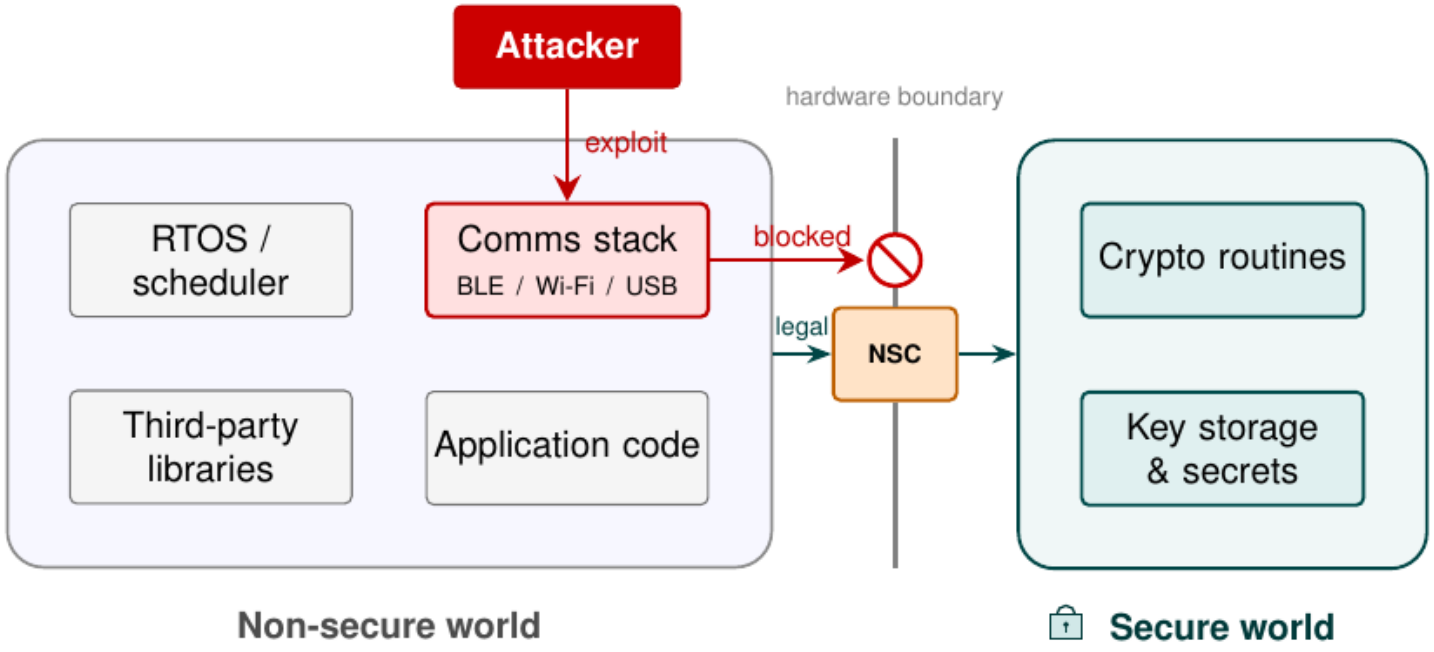
- The CPU is in one state at a time
- Non-secure code touches secure memory? Fault.
- The check happens in hardware, not in the OS



https://inet.haw-hamburg.de/thesis/completed/ma_lena_boeckmann.pdf/view

Motivation

With TrustZone-M:
The same exploit still compromises the non secure world
The hardware boundary stops it at the edge of the secure world
Keys cannot be extracted and cryptoroutines can’t be called
Only through the NSC gateway can the secure world be accessed



A secure home for our keys

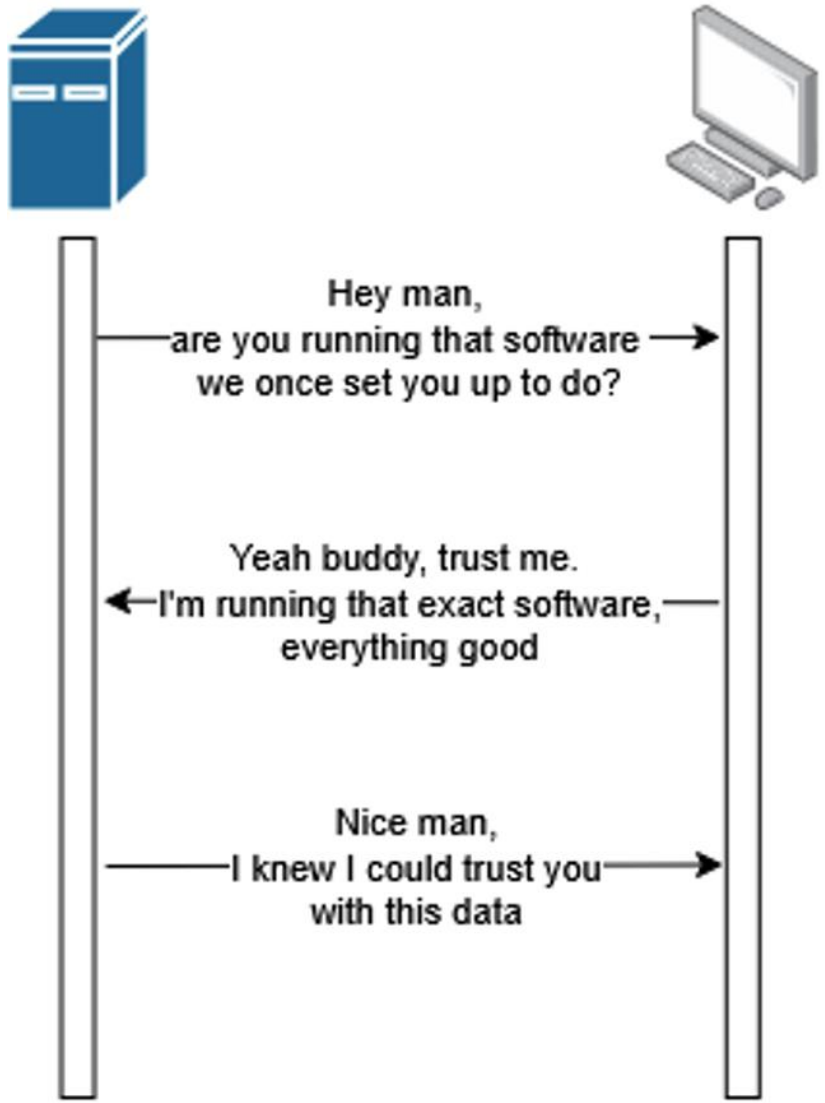
- We want a place that:
  - Is secured against physical tampering / flash extraction
  - Software bugs / flash dumps from the inside

- We can achieve that by using:
  - A hardware keystore
  - An isolation mechanism like TrustZone-M

Remote Attestation

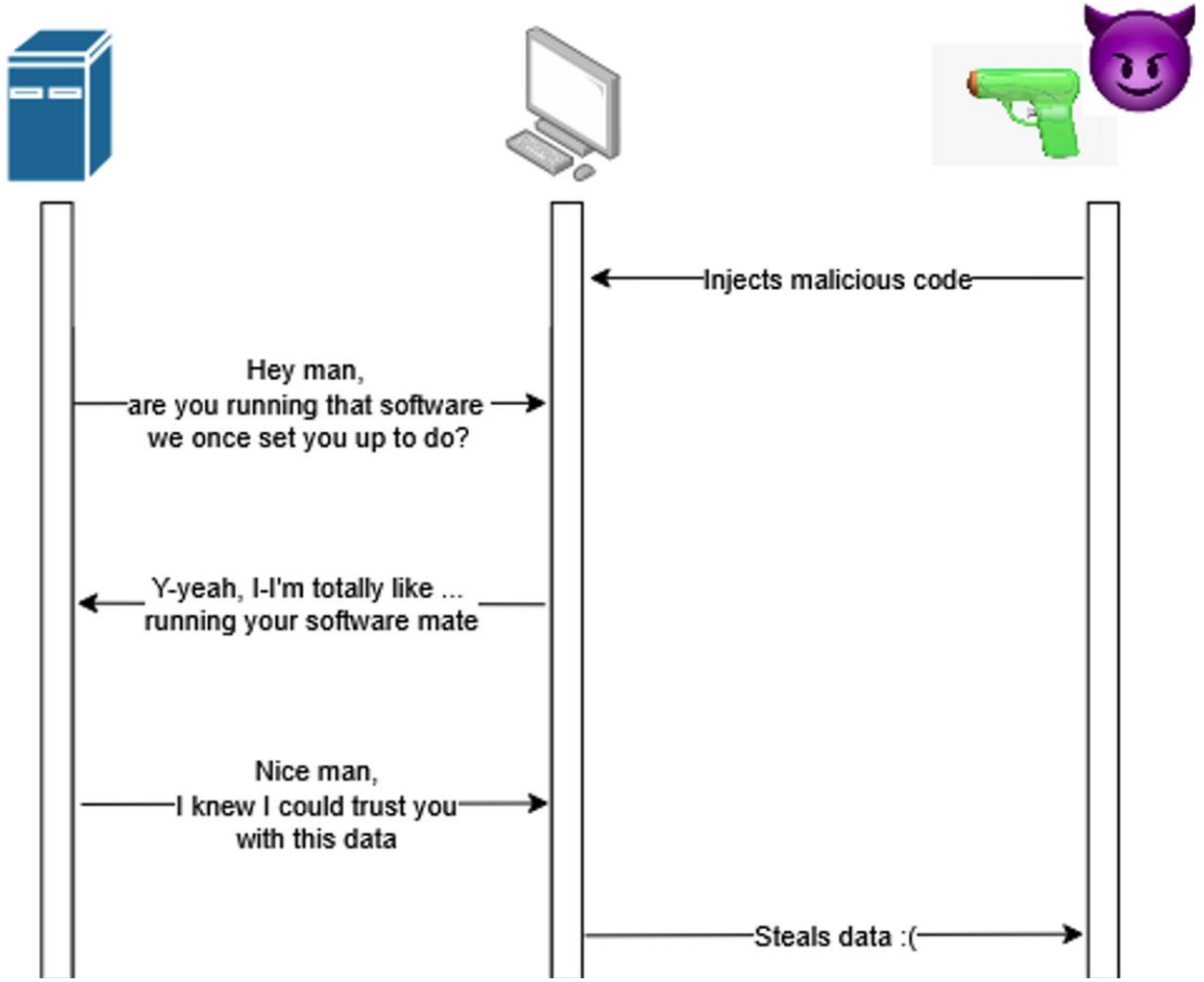
How we can communicate the systems state to the outside world

Problem statement



Problem statement

The server asks.
The device answers.
A compromised device gives the same answer as a healthy one

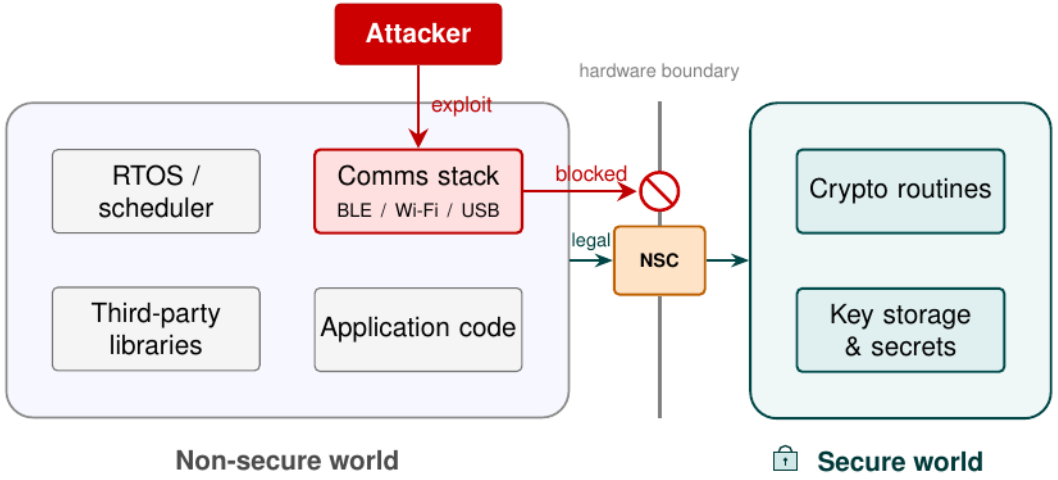


Problem statement

- The answer must not come from the part that can be compromised
- It must be about something the server can check
- It must be about now, not about last year
- It must be about this exact device

Problem statement - requirements

The OS passes the question through. It does not write the answer
Compromising RIOT lets you request an answer. It does not let you author one



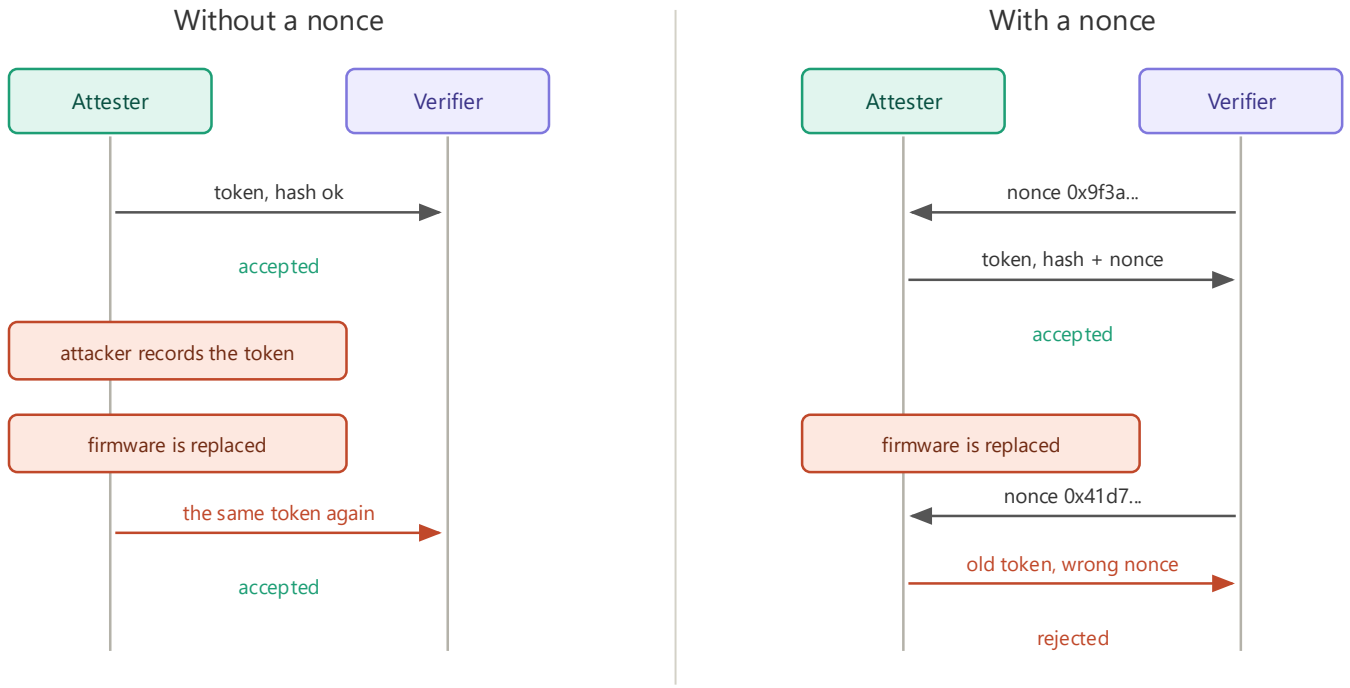
With TrustZone-M: the crypto routines and keys stay secure. The same exploit still compromises the non-secure world, but the hardware boundary stops it at the edge of the secure world – it cannot read the keys or call into the crypto routines. Legitimate non-secure code reaches them only through the NSC gateway.

Problem statement - requirements

A hash of the image, taken before the image ran
The server compares it with the hash it expects

Problem statement - freshness

An old answer verifies just as well as a new one
So the server picks a random value and sends it with the question
The device folds it into the answer

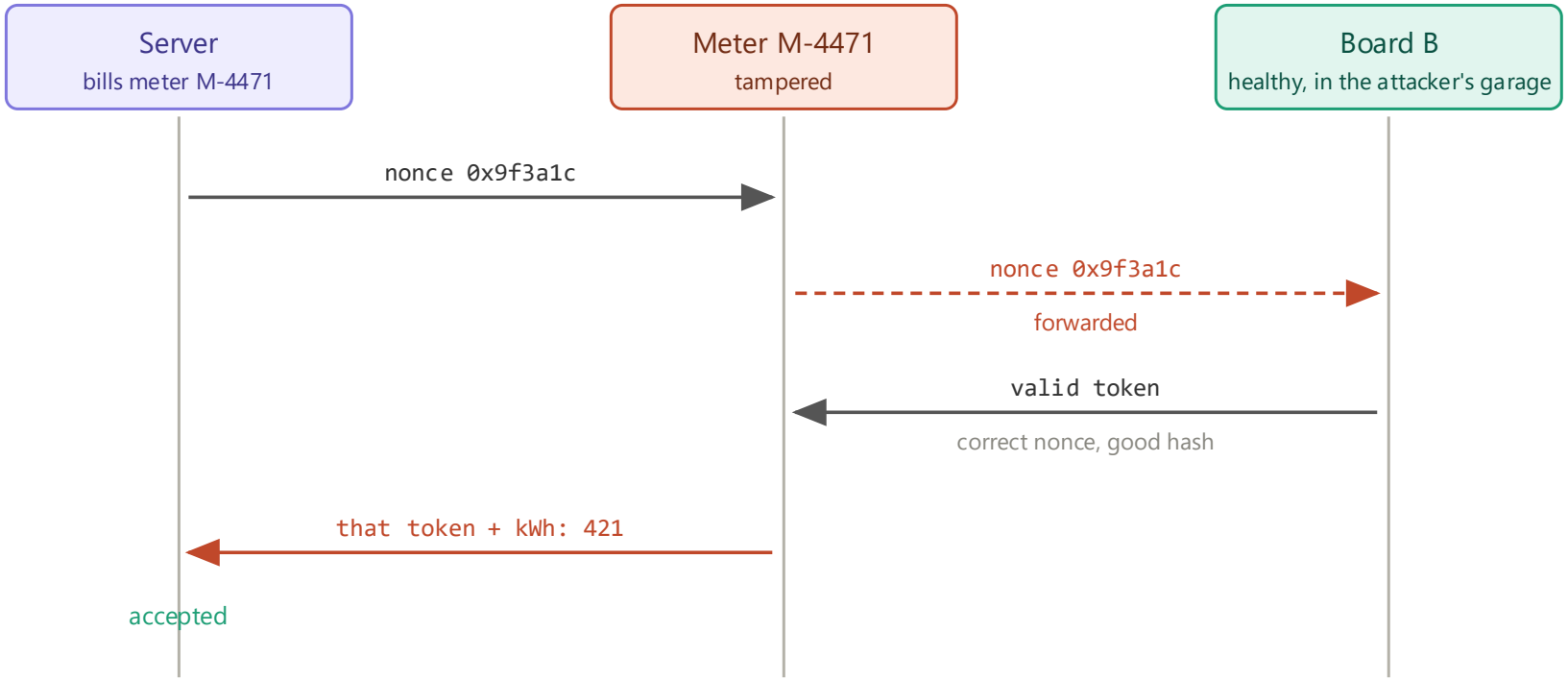


The verifier decides what counts as recent.

Problem statement - identiy

- The answer proves that some device booted a good image
- It does not say which one
- Buy a second board, keep it healthy, let it answer for the broken one
- So the statement has to name the unit it is about
- And someone has to vouch that the key belongs to that unit

Problem statement - identiy



Every check passes. The token just never said which device it came from.

Problem statement - in a nutshell

- The answer has to come from behind the boundary
- It has to carry something checkable, not a yes
- It has to be recent
- It has to name the unit
- Someone has to vouch for the key

Remote Attestation - rfc9334

Status:	Informational
More info:	Errata exist | Datatracker| IPR | Info page

Stream:	Internet Engineering Task Force (IETF)
RFC:	9334
Category:	Informational
Published:	January 2023
ISSN:	2070-1721

Authors:	H. Birkholz	D. Thaler	M. Richardson	N. Smith	W. Pan
	Fraunhofer SIT	*Microsoft*	*Sandelman Software Works*	*Intel*	*Huawei*

RFC 9334
Remote ATtestation procedureS (RATS) Architecture

Abstract

In network protocol exchanges, it is often useful for one end of a communication to know whether the other end is in an intended operating state. This document provides an architectural overview of the entities involved that make such tests possible through the process of generating, conveying, and evaluating evidentiary Claims. It provides a model that is neutral toward processor architectures, the content of Claims, and protocols.

Remote Attestation - rfc9334 mapping

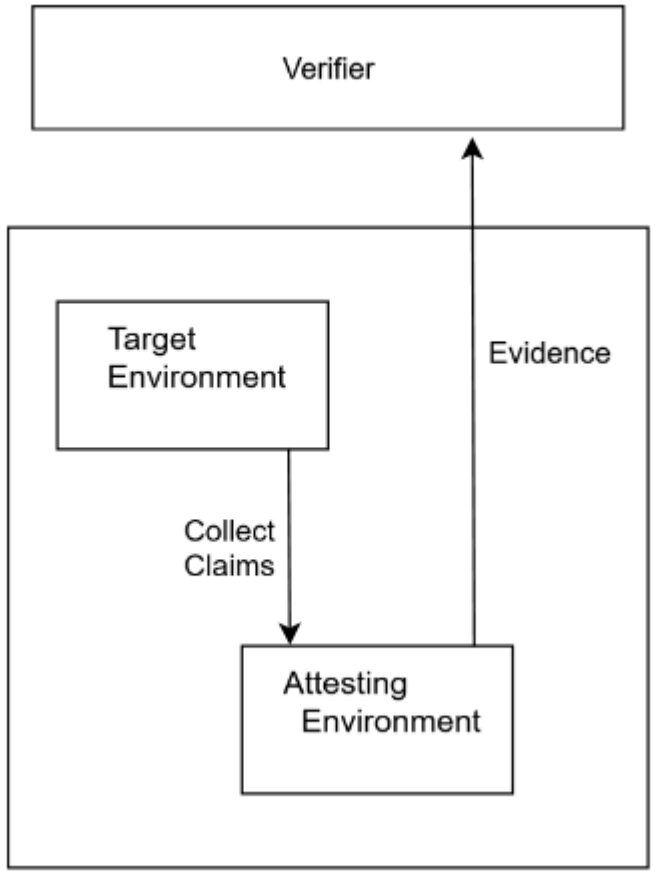
what we worked out	what RFC 9334 calls it
the part that answers, behind the boundary	Attesting Environment
the part it describes	Target Environment
the signed statement it sends	Evidence
the hash the server expects to see	Reference Values
someone vouching that the key is this unit	Endorsement
whoever checks all of that	Verifier
whoever acts on the outcome	Relying Party

Remote Attestation - rfc9334

RATS Architecture:
Have the Attesting environment be inside the TEE

Have the main app / RIOT OS be in the Target environment

Have the Attesting environment collect and sign claims into evidence. Then report that evidence to the Verifier

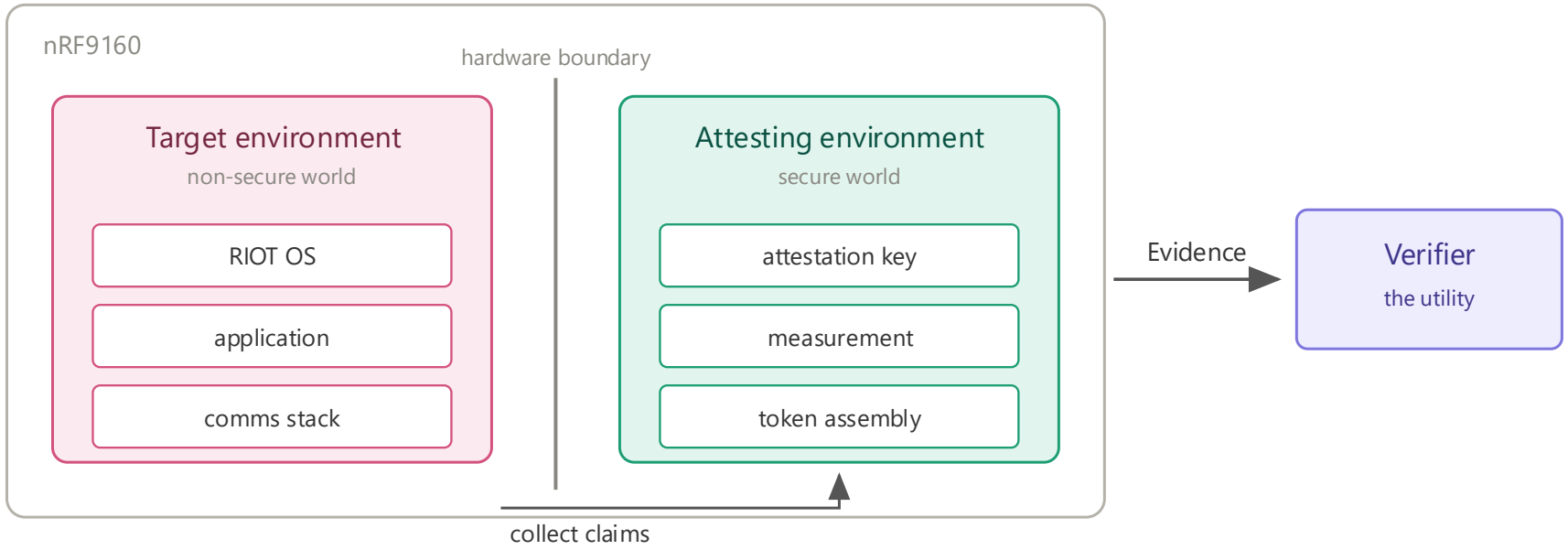


https://datatracker.ietf.org/doc/html/rfc9334

Our contribution

What we built

Remote Attestation - what I built

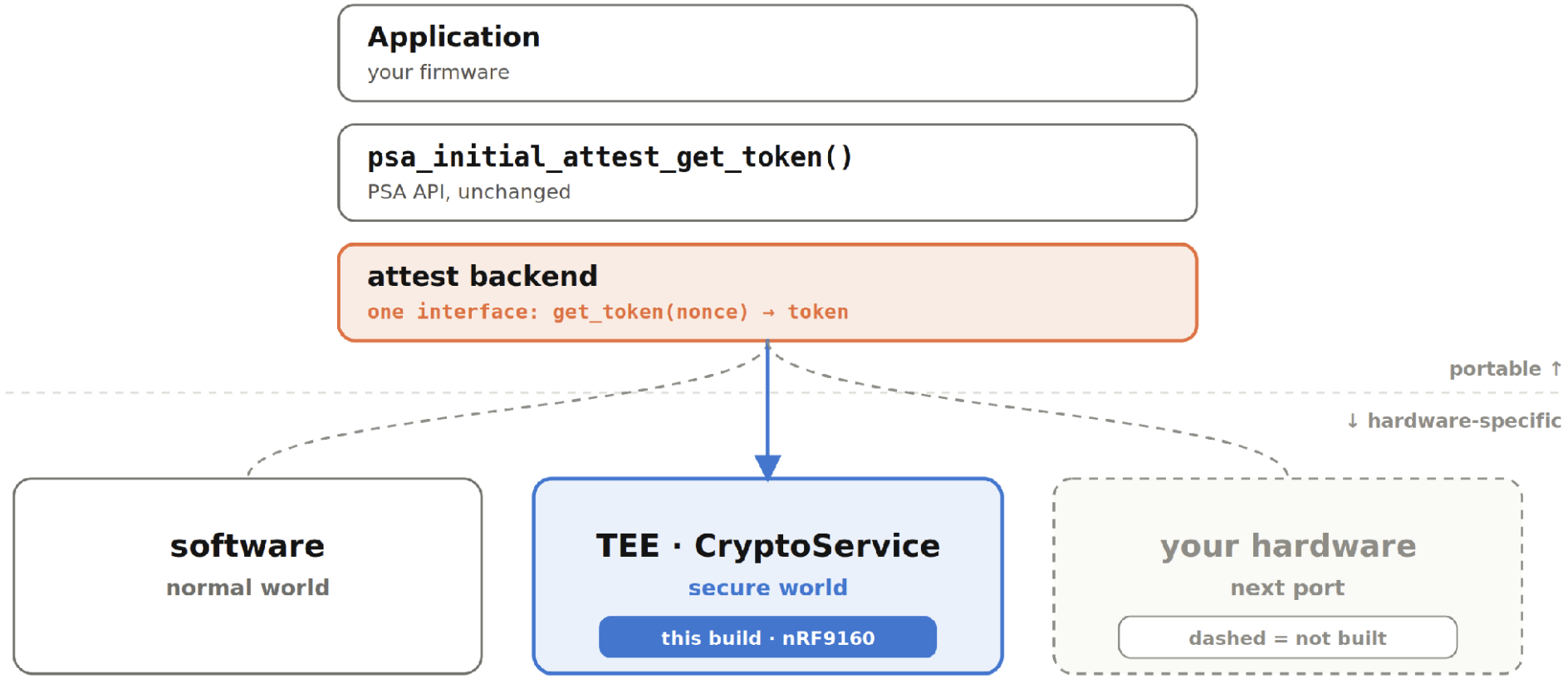


The environment being described is not the environment doing the describing.

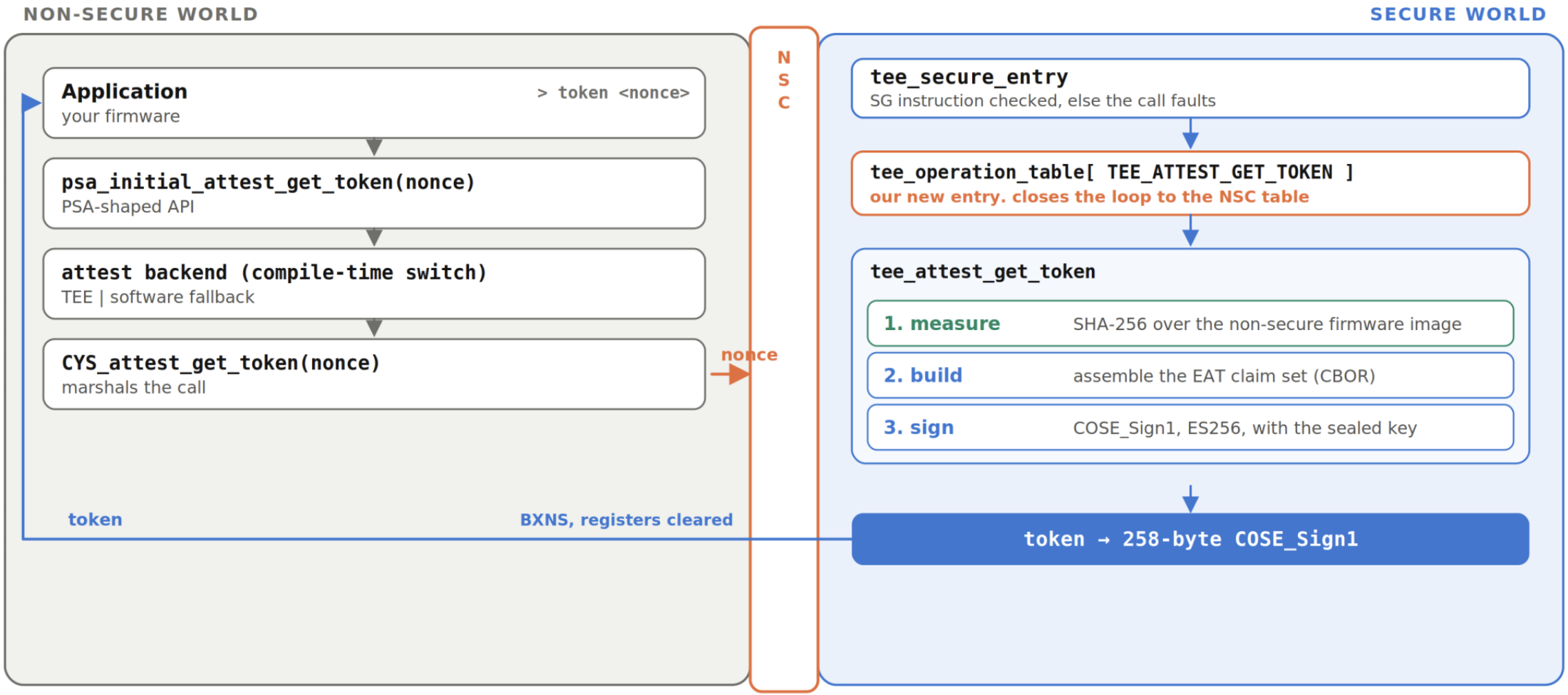
Our contribution - what we built

- riot-tee for nrf9160 already existed (Lena Boeckmann). Secure world, keys, crypto, NSC door
- CryptoServiceAPI already existed (Lars Pfau)
- We added attestation on top: measure, build the claim set, sign
- PSA-shaped API on the non-secure side
- Two backends, picked at compile time
- Follows RFC 9334, token per RFC 9711

Our contribution - core architecture



Our contribution - hardware backend



Our contribution - hardware backend

```
psa_status_t attest_backend_get_token(psa_key_id_t iak,
                                      const uint8_t *nonce,
                                              size_t nonce_len,
                                      uint8_t *token,
                                                              size_t      token_size,
                                                                                      size_t
*token_len);
```

Our contribution - hardware backend

```
CYS_error_t tee_attest_get_token(const sealed_key_t *sealed,
                        const uint8_t *nonce, size_t nonce_len,
                        uint8_t *token, size_t *token_len)
{
   uint8_t measurement[32], priv[32], pubkey[65];
   uint8_t instance_id[33], payload[512];
   size_t  payload_len;

   /* 1. measure: hash the non-secure firmware image */
   tee_sha256(FLASH_START_NS, NS_FLASH_SIZE, measurement);

   /* 2. unseal the attestation key (only possible in here) */
   tee_rot_decrypt_key_ocb(sealed, priv);

   /* 3. identity = 0x01 || SHA-256(public key) */
   tee_ecc_p256_derive_pubkey(priv, pubkey);
   instance_id[0] = 0x01;
   tee_sha256(pubkey, sizeof pubkey, instance_id + 1);

   /* 4. build the EAT claims: nonce + identity + measurement */
   encode_claims(payload, sizeof payload, nonce, nonce_len,
            instance_id, measurement, &payload_len);

   /* 5. sign it all as a COSE_Sign1 */
   return sign_cose(payload, payload_len, priv, token, token_len);
}
```

Our contribution - the token

Our contribution - cost

WHAT WE MEASURED	COST
Flash added by attestation the EAT and COSE builder, 4 KB of the 128 KB secure partition	+4 KB
Static RAM added nothing added to .bss or .data	~0 B
Peak stack per token transient buffers, two 512 B CBOR buffers dominate, freed on return	~1.3 KB
Non-secure app growth only the PSA call site is added to the normal world	~0.1 KB
Token size one COSE_Sign1 over the EAT claim set	258 B
Latency per token average of 100 runs, 0.92 s to 1.00 s, dominated by hashing 896 KB of flash	0.94 s

Weaknesses

Time to attack

Problem statement - what we built

- You are the attacker. The device tells the verifier: "I run genuine firmware."
- You get: the network,code in the normal world, the board in your hand
- How do you make the verifier trust a device you tampered with?

Problem statement - what we built

- Self-made key, how do we handle that?
- Hashes stored firmware, not the running system
- Software backend: key in plain memory
- No secure boot, the TEE is trusted blindly
- Physical access: read the key or glitch the check
- Unlocked debug port!!!

In a nutshell

What you should take away from this talk

Problem statement - what I built

- A device proves what software it is running to a remote verifier.
- The proof is signed evidence: a measurement of its firmware, plus the verifier's fresh nonce.
- Hard to forge, because the signing key never leaves the hardware.
- The verifier checks the evidence and decides whether to trust the device.

Discussion

Your questions and ideas

Backup

Initialization

Real world code:
This code is part of the SW Firmware image. It sets up different registers and configures the secure world.

At the end, it sets the MSP and yields to the non secure main

main_s.c

```c
/* Secure-world startup — nRF9160 (Cortex-M33, TrustZone-M).
 * Runs first out of reset: partitions the SoC, brings up the
 * root of trust, then branches into the non-secure image. */
#include <stdint.h>
#include <arm_cmse.h>
#include "nrf_spu.h"
#include "tee_rot.h"
/* Start of the non-secure image (from the NS linker script) */
extern uint32_t FLASH_START_NS[];
/* A call into non-secure code — the compiler emits a BLXNS */
typedef void __attribute__((cmse_nonsecure_call)) ns_entry_t(void);
/* Peripherals we hand to the non-secure world */
static const uint32_t ns_peripherals[] = {
    NRFX_PERIPHERAL_ID_GET(NRF_P0_NS),      /* GPIO   */
    NRFX_PERIPHERAL_ID_GET(NRF_UARTE0_NS),  /* UART   */
    NRFX_PERIPHERAL_ID_GET(NRF_TIMER0_NS),  /* timers */
    NRFX_PERIPHERAL_ID_GET(NRF_TIMER1_NS),
};
int main(void)
{
    /* 1. Partition memory with the SPU.
     *    Flash: 32 x 32 KB, first 4 regions  (128 KB) stay secure.
     *    RAM:   32 x  8 KB, first 11 regions (88 KB)  stay secure. */
    for (uint32_t i = 4; i < 32; i++) {
        NRF_SPU_S->FLASHREGION[i].PERM &= ~SPU_FLASHREGION_PERM_SECATTR_Msk;
    }
    for (uint32_t i = 11; i < 32; i++) {
        NRF_SPU_S->RAMREGION[i].PERM &= ~SPU_RAMREGION_PERM_SECATTR_Msk;
    }
    /* 2. Carve out the Non-Secure Callable (NSC) region — the only
     *    entry point into secure code. 32 B at the top of the last
     *    secure flash region (region 3). */
    NRF_SPU_S->FLASHNSC[0].REGION = 3;
    NRF_SPU_S->FLASHNSC[0].SIZE   = NRF_SPU_NSC_SIZE_32B;
    /* 3. Hand the selected peripherals to the non-secure world. */
    for (uint32_t i = 0; i < ARRAY_SIZE(ns_peripherals); i++) {
        NRF_SPU_S->PERIPHID[ns_peripherals[i]].PERM &= ~SPU_PERIPHID_PERM_SECATTR_Msk;
    }
    /* 4. Secure-only work: start the crypto engine and make sure the
     *    platform AES key exists. The key lives in secure RAM and is
     *    never visible to the non-secure side. */
    tee_init_random();
    cc3xx_lowlevel_init();
    tee_rot_try_generate_aes_key();
    /* 5. Enter the non-secure image: point the NS vector table at it,
     *    load the NS main stack pointer, branch to its reset handler. */
    SCB_NS->VTOR = (uint32_t)FLASH_START_NS;
    __TZ_set_MSP_NS(FLASH_START_NS[0]);
    ns_entry_t *ns_reset = (ns_entry_t *)FLASH_START_NS[1];
    ns_reset();
    while (1) { }  /* never reached */
}
```

Initialization

```c
/* 1. Partition memory with the SPU.
 *     Flash: 32 x 32 KB, first 4 regions  (128 KB) stay secure.
 *     RAM:   32 x  8 KB, first 11 regions (88 KB)  stay secure. */
for (uint32_t i = 4; i < 32; i++) {
    NRF_SPU_S->FLASHREGION[i].PERM &= ~SPU_FLASHREGION_PERM_SECATTR_Msk;
}
for (uint32_t i = 11; i < 32; i++) {
    NRF_SPU_S->RAMREGION[i].PERM &= ~SPU_RAMREGION_PERM_SECATTR_Msk;
}
```

main_s.c

```c
/* Secure-world startup — nRF9160 (Cortex-M33, TrustZone-M).
 * Runs first out of reset: partitions the SoC, brings up the
 * root of trust, then branches into the non-secure image. */
#include <stdint.h>
#include <arm_cmse.h>
#include "nrf_spu.h"
#include "tee_rot.h"
/* Start of the non-secure image (from the NS linker script) */
extern uint32_t FLASH_START_NS[];
/* A call into non-secure code — the compiler emits a BLXNS */
typedef void __attribute__((cmse_nonsecure_call)) ns_entry_t(void);
/* Peripherals we hand to the non-secure world */
static const uint32_t ns_peripherals[] = {
    NRFX_PERIPHERAL_ID_GET(NRF_P0_NS),      /* GPIO   */
    NRFX_PERIPHERAL_ID_GET(NRF_UARTE0_NS),  /* UART   */
    NRFX_PERIPHERAL_ID_GET(NRF_TIMER0_NS),  /* timers */
};
int main(void)
{
    /* 1. Partition memory with the SPU.
     *    Flash: 32 x 32 KB, first 4 regions  (128 KB) stay secure.
     *    RAM:   32 x  8 KB, first 11 regions (88 KB)  stay secure. */
    for (uint32_t i = 4; i < 32; i++) {
        NRF_SPU_S->FLASHREGION[i].PERM &= ~SPU_FLASHREGION_PERM_SECATTR_Msk;
    }
    for (uint32_t i = 11; i < 32; i++) {
        NRF_SPU_S->RAMREGION[i].PERM &= ~SPU_RAMREGION_PERM_SECATTR_Msk;
    }
    /* 2. Mark the peripherals we hand to the non-secure world as non-secure. */
    for (uint32_t i = 0; i < sizeof(ns_peripherals)/sizeof(ns_peripherals[0]); i++) {
        NRF_SPU_S->PERIPHID[ns_peripherals[i]].PERM &= ~SPU_PERIPHID_PERM_SECATTR_Msk;
    }
    /* 4. Secure-only work: start the crypto engine and make sure the
     *    platform AES key exists. The key lives in secure RAM and is
     *    never visible to the non-secure side. */
    tee_init_random();
    cc3xx_lowlevel_init();
    tee_rot_try_generate_aes_key();
    /* 5. Enter the non-secure image: point the NS vector table at it,
     *    load the NS main stack pointer, branch to its reset handler. */
    SCB_NS->VTOR = (uint32_t)FLASH_START_NS;
    __TZ_set_MSP_NS(FLASH_START_NS[0]);
    ns_entry_t *ns_reset = (ns_entry_t *)FLASH_START_NS[1];
    ns_reset();
    while (1) { }  /* never reached */
}
```

Secure World Entry

NSC Gateway:
There exists only a single tee_secure_entry function

Secure world routines:
NSC callable routines are registered in the tee_operation_table

non_secure_entry.c

```c
/* Secure entry veneer — placed in the NSC region.
 * cmse_nonsecure_entry makes it callable from the non-secure
 * world (the compiler emits the SG instruction). It dispatches
 * to one secure operation from the table. */
#include <arm_cmse.h>
#include "tee_operation_table.h"

__attribute__((cmse_nonsecure_entry))
CYS_error_t tee_secure_entry(io_operation_info_t *op_info,
                             io_pack_in_t *in,
                             io_pack_out_t *out)
{
    tee_operation_t op = tee_operation_table[op_info->operation];
    if (op == NULL) {
        return CYS_ERROR_NOT_SUPPORTED;
    }
    return op(in, op_info->in_len, out, op_info->out_len);
}
```

tee_operation_table.h

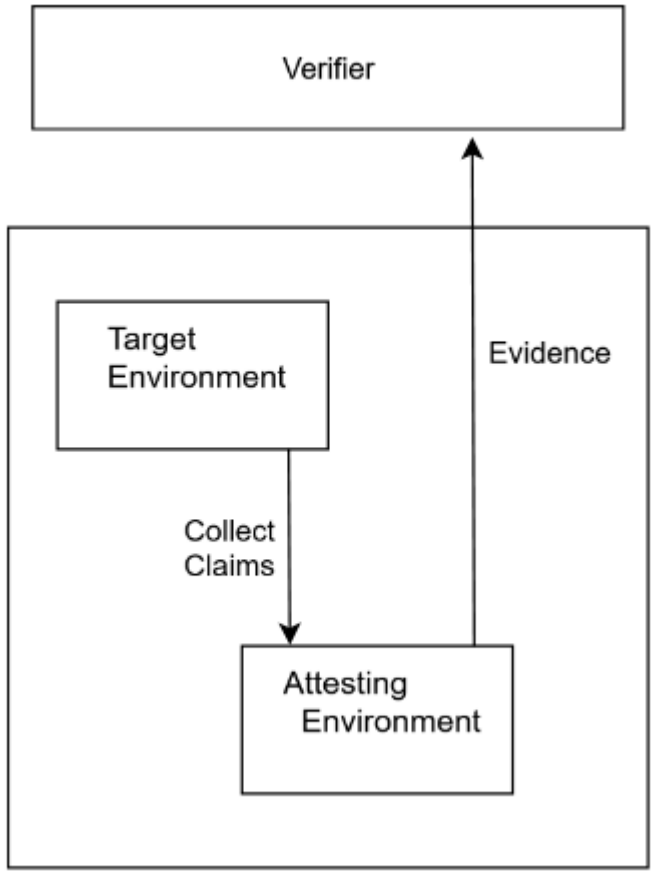
```c
/* Table of secure operations the non-secure world may invoke.
 * Each entry maps an operation ID to its secure handler. */
#ifndef TEE_OPERATION_TABLE_H
#define TEE_OPERATION_TABLE_H

#include "CYS/common.h"
#include "tee_operations.h"

typedef CYS_error_t (*tee_operation_t)(io_pack_in_t *in,  size_t in_len,
                                       io_pack_out_t *out, size_t out_len);

static const tee_operation_t tee_operation_table[] = {
    [TEE_RANDOM_GENERATE]            = tee_generate_random_bytes,
    [TEE_HASH_SHA256_UPDATE]         = tee_hash_sha256_update,
    [TEE_CIPHER_AES_128_CBC_ENCRYPT] = tee_cipher_aes_128_cbc_encrypt,
    [TEE_CIPHER_AES_128_CBC_DECRYPT] = tee_cipher_aes_128_cbc_decrypt,
    [TEE_ECC_P256_SIGN_HASH]         = tee_ecc_p256_sign_hash,
};

#endif /* TEE_OPERATION_TABLE_H */
```

Remote Attestation

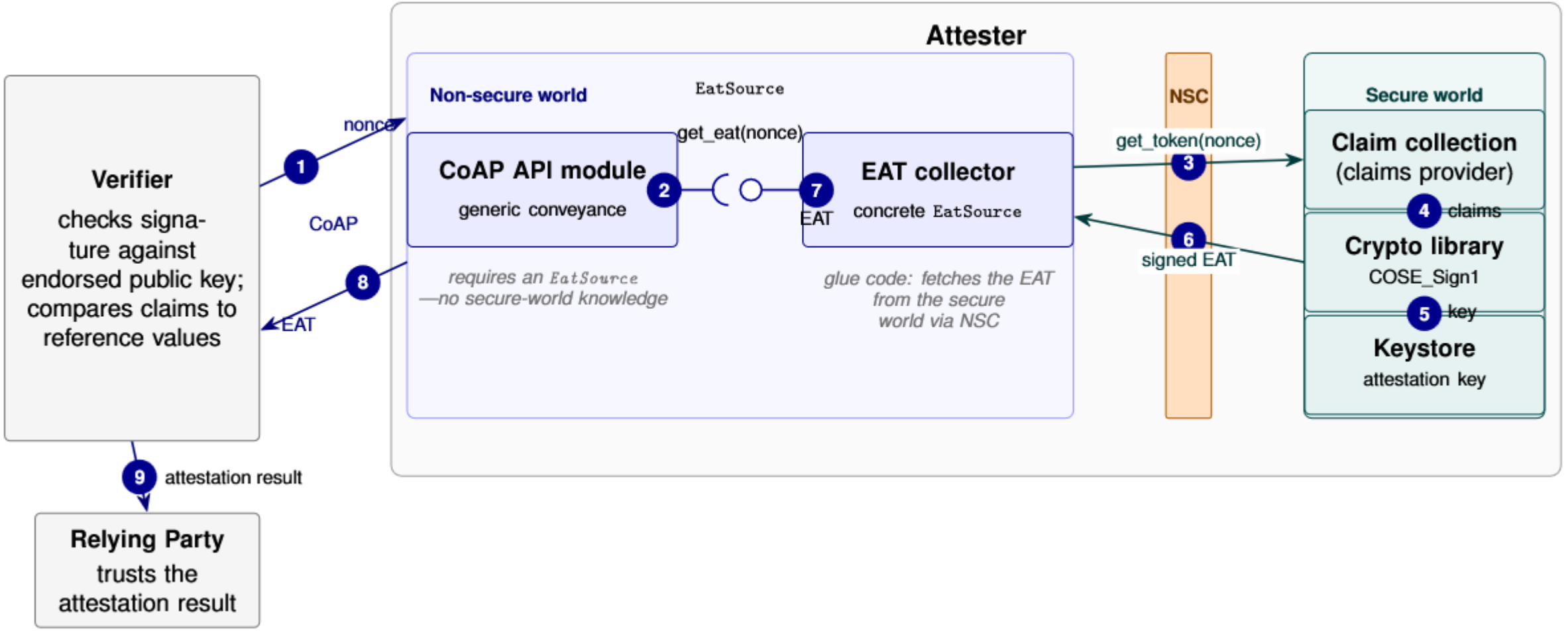
RATS Architecture:
Have the Attesting environment be inside the TEE

Have the main app / RIOT OS be in the Target environment

Have the Attesting environment collect and sign claims into evidence. Then report that evidence to the Verifier



Remote Attestation



Remote attestation vs certificates

Certificates attest identity

Remote attestation attests state