SCML/16

SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

Implementing Crypto Security in Bare-Metal Cortex-M

Jonny Doin – GridVortex

#SCML2016

First, a little about me

Disclaimer: I am NOT a cryptographer.

I design and implement Embedded Systems for critical applications.

Along the years I was deeply involved with Hardware Design, Analog Electronics, Chip Design, Firmware, Networking, Security.

I see Embedded Design as a continuum that encompasses all of the above, in a harmonious System, targeted to specific functionalities.

That is a very useful vision when implementing Embedded Security.

Agenda

- Motivations
- Requirements for Security Systems: NIST FIPS-140-2
- Standard documents: NIST and RFCs
- Design decisions
- Implementation details: SHA-256, HMAC, (N)RBG
- Digital Signatures, keys and hardware acceleration
- RAM secureFence
- Tamper and Fraud detection

http://www.sourcesecurity.com/images/moreimages/PCSC-fault-tolerance-250.jpg

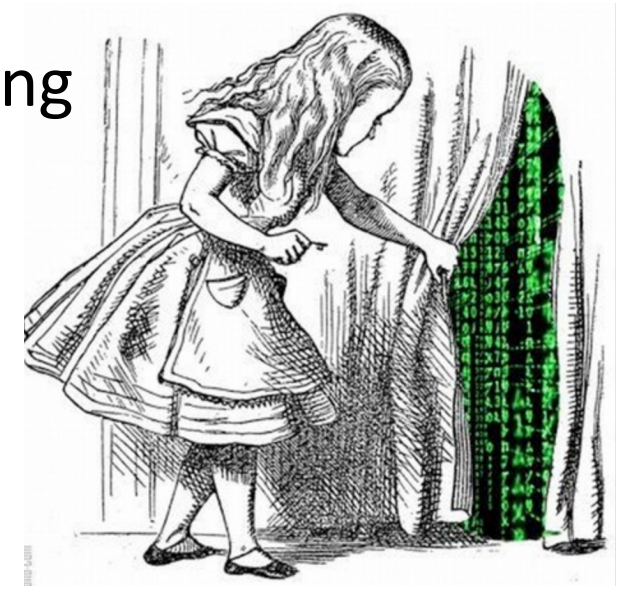
Motivations

At GridVortex, we decided to write our Security Library from scratch.

Embedded Security is an important aspect of all the systems we are designing right now, including IoT chip design.

We think the SmartCity cannot happen without strong Embedded Security.

This presentation is a walkthrough on the design and implementation of our Embedded Security Lib for Cortex-M, with some detail on the how-to implement some essential blocks.



http://cfbmatrix.com/wp-content/uploads/2013/11/alice-in-matrix.jpg

Requirements: Legal Metrology

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

- Software requirements for Legal Metrology systems are emerging
- Demand for anti-fraud on smart meters, fuel dispensers and weight scales
- Strong Digital Signature support
- Firmware identification and verification
- Tamper detection and contention
- Source-code Certification on the Embedded Firmware

Requirements: NIST FIPS-140-2

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

- Specifies the minimum requirements for cryptographic modules
- Provides certification recommendations for 4 levels of security
- We want to certify systems at Levels 2 and 3
- All modules shall:
  - Show Security Status
  - Perform Self-Tests
  - Perform Approved Security Functions
  - Tamper detection and response
- User authentication is required to access secrets and configuration

FIPS PUB 140-2
CHANGE NOTICES (12-03-2002)

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION
(Supercedes FIPS PUB 140-1, 1994 January 11)

SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES

CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

Issued May 25, 2001

U.S. Department of Commerce
Donald L. Evans, Secretary

Technology Administration
Phillip J. Bond, Under Secretary for Technology

National Institute of Standards and Technology
Arden L. Bement, Jr., Director

SIBAPEM

INMETRO

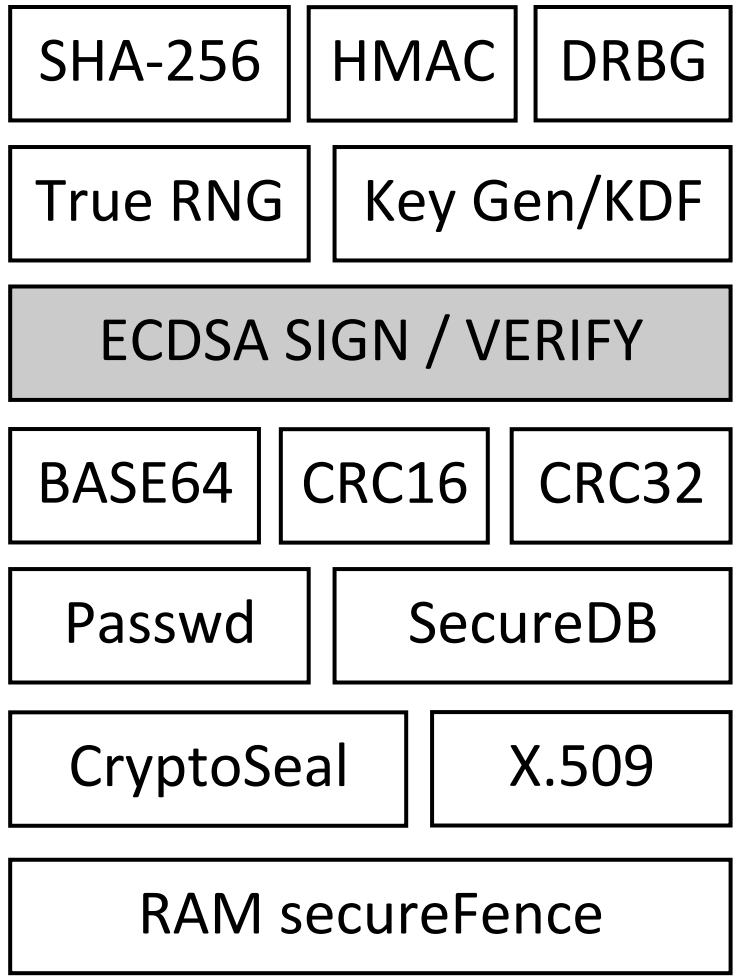
Standard Docs: NIST and RFCs

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

- Cryptography is NOT a secret
- Strong cryptography is a technology available to everyone
- All algorithms and constructions are FREE published standards
- You can download them from NIST publications and IETF RFCs
  - Algorithms, Implementation Recommendations, Test Vectors
- Every implementation needs to follow these requirements and docs

Design Decisions: GV_Securelib

- Simple, Lightweight, *proper*, support for Security
- Optimized for ARM-Cortex-M
- Hybrid Implementation:
  - Fast Firmware for base functions
  - Hardware Acceleration for expensive functions
  - Key storage uses dedicated Hardware
  - Services with State-Machine friendly callback messages
- Support for standard data encodings
- High Reliability (full fault detection)



Implementing: SHA-256

SHA-256 is a block transformation function that operates on 512bit blocks, and generates a 256bit output register.

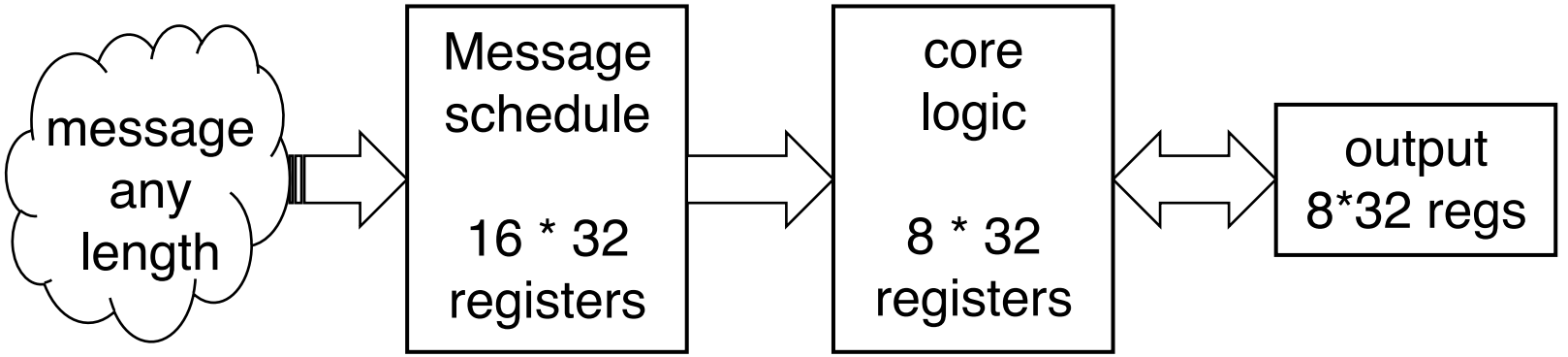
For a silicon implementation, SHA-256 is usually designed as a pipelined register machine, with 256-wide parallel logic, that produces one block hash on ~68 clocks, depending on the implementation.

It has 3 large logic blocks:

the *message schedule*

the *block hash core*

the *output register*



Implementing: SHA-256 (1)

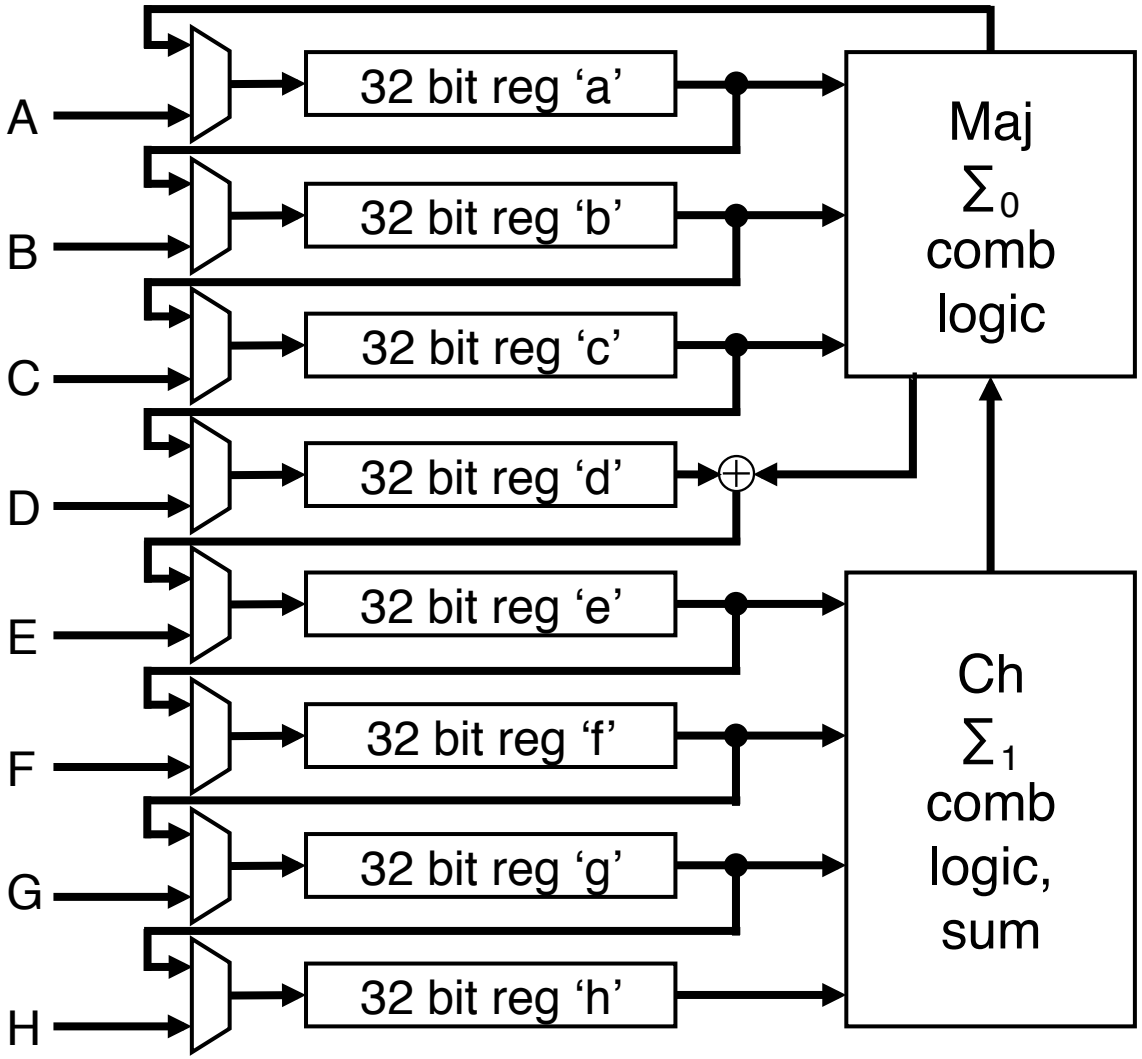
The core block is where the action is. (this is a simplified schematic of the logic)

This block has 8 registers and the ARX (Add-Rotate-Xor) functions.

It takes 66 clock cycles to compute a hash block of 64 bytes.

{A..H} are the output registers.

After 64 clocks, {A..H} += {a..h}



Implementing: SHA-256 (2)

ARM Architectural traits related to SHA-256 logic:

- 16 CPU registers (can hold the 8 core registers in the CPU)
- Barrel Shifter coupled to ALU (multiple operations per instruction)

```
EOR     r10,r10,r6,ROR #11 ; r10 ^= (r6 >> 11 | ((r6 << 21) & 0xFFFFFFFF))
```

- Byte-reverse operations (Can work on big-endian data)

```
REV     r6,r6 ; ntoh(r6)
```

We can write SHA-256 with minimum RAM access for the core logic, and minimized stack footprint.

By reducing memory accesses, we improve both speed and security.

Implementing: SHA-256 (3)

With these aspects in mind, we implemented the core logic with:

- 100 stack bytes, instead of the original 288 stack bytes
- Zero memory access for the {a..h} registers
- Only one access per permutation for the Ki constants
- 3 loads and 1 store for the Wt array
- Zero-stack footprint for the H[0..7] array
- Zero-copy logic for the whole core
- Fused algorithm for the Schedule and Core, resulting in a single pass

The resulting SHA-256 takes less than 25us per 64-byte block for a complete hash computation, comparing very favorably to x86 software implementations.

Implementing: SHA-256 (4)

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

Details:

```
{
    a = hctx->H[0];
    ...
    h = hctx->H[7];

    ARX(a, b, c, d, e, f, g, h, Wm( 0), 0x428A2F98);
    ARX(h, a, b, c, d, e, f, g, Wm( 1), 0x71374491);
    ...
    ARX(c, d, e, f, g, h, a, b, Wt(62), 0xBEF9A3F7);
    ARX(b, c, d, e, f, g, h, a, Wt(63), 0xC67178F2);

    hctx->H[0] += a;
    ...
    hctx->H[7] += h;
}
```

Each line ARX macro is one ‘clock’ of the core logic

Schedule and Internal arrays scanned in one pass

Registers are ‘shifted’ through the ARX network

Implementing: SHA-256 (5)

The ARX network macros:

```
#define s0(x)   (RRX((x), 7) ^ RRX((x),18) ^ SHR((x), 3))
#define s1(x)   (RRX((x),17) ^ RRX((x),19) ^ SHR((x),10))
#define SIG0(x) (RRX((x), 2) ^ RRX((x),13) ^ RRX((x),22))
#define SIG1(x) (RRX((x), 6) ^ RRX((x),11) ^ RRX((x),25))
#define Ch(x,y,z)  ((x & y) + ((~x) & z))
#define Maj(x,y,z) ((y & (x ^ z)) + (x & z))
#define Wm(t) (W[t] = REV(M[t]))
#define Wt(t) (W[t & 0xF] += s0(W[(t + 1) & 0xF]) + s1(W[(t + 14) & 0xF]) + W[(t + 9) & 0xF])

#define ARX(a,b,c,d,e,f,g,h,Wi,Ki)                  \
{                                                   \
    uint32_t T = h + Ch(e,f,g) + SIG1(e) + Wi + Ki; \
    d += T;                                         \
    h = T + Maj(a,b,c) + SIG0(a);                   \
}
```

Implementing: SHA-256 (6)

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

One ARX iteration:

```
Ch    0x0003E820 EA070A08 AND   r10,r7,r8
      0x0003E824 EA240B07 BIC   r11,r4,r7
      0x0003E828 44DA     ADD   r10,r10,r11
      0x0003E82A 4456     ADD   r6,r6,r10
SIG1  0x0003E82C EA4F1AB7 ROR   r10,r7,#6
      0x0003E830 EA8A2AF7 EOR   r10,r10,r7,ROR #11
      0x0003E834 EA8A6A77 EOR   r10,r10,r7,ROR #25
      0x0003E838 4456     ADD   r6,r6,r10
s0    0x0003E83A F8DDA02C LDR   r10,[sp,#0x2C]
      0x0003E83E EA4F1BFA ROR   r11,r10,#7
      0x0003E842 EA8B4BBA EOR   r11,r11,r10,ROR #18
      0x0003E846 EA8B0BDA EOR   r11,r11,r10,LSR #3
s1    0x0003E84A F8DDA028 LDR   r10,[sp,#0x28]
      0x0003E84E EA4F4E7A ROR   lr,r10,#17
      0x0003E852 EA8E4EFA EOR   lr,lr,r10,ROR #19
      0x0003E856 EA8E2A9A EOR   r10,lr,r10,LSR #10
      0x0003E85A EB0B0E0A ADD   lr,r11,r10
Wt    0x0003E85E F8DDB04C LDR   r11,[sp,#0x4C]
      0x0003E862 F8DDA030 LDR   r10,[sp,#0x30]
      0x0003E866 44DA     ADD   r10,r10,r11
      0x0003E868 44F2     ADD   r10,r10,lr
      0x0003E86A 4456     ADD   r6,r6,r10
      0x0003E86C F8CDA030 STR   r10,[sp,#0x30]
Ki    0x0003E870 F8DFA220 LDR.W r10,[pc,#544]
Maj   0x0003E874 EA030B01 AND   r11,r3,r1
      0x0003E878 44B2     ADD   r10,r10,r6
      0x0003E87A EB02060A ADD   r6,r2,r10
      0x0003E87E EA830201 EOR   r2,r3,r1
      0x0003E882 EA02020C AND   r2,r2,r12
      0x0003E886 445A     ADD   r2,r2,r11
      0x0003E888 4452     ADD   r2,r2,r10
SIG0  0x0003E88A EA4F0AB3 ROR   r10,r3,#2
      0x0003E88E EA8A3A73 EOR   r10,r10,r3,ROR #13
      0x0003E892 EA8A5AB3 EOR   r10,r10,r3,ROR #22
      0x0003E896 4452     ADD   r2,r2,r10
```

SIBAPEM

INMETRO

Implementing: SHA-256 (7)

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

Lib Interface:

```
typedef struct hash_context_t {
    uint32_t bytecount;          //!< total message length (2**29 bytes = 536870912 bytes)
    volatile uint32_t H[8];      //!< intermediate hash value H(i)
    uint8_t W[64];               //!< 512bits hash data block
    uint8_t ipad[64];            //!< inner padding for HMAC-SHA-256
    uint8_t opad[64];            //!< outer padding for HMAC-SHA-256
} HASH_CONTEXT_T;

const char *str_secure_status(SECURE_STATUS_T err_code);
SECURE_STATUS_T sha256_init(HASH_CONTEXT_T *hctx);                                  //!< Initialize context
SECURE_STATUS_T sha256_uninit(HASH_CONTEXT_T *hctx);                                //!< Clear SHA-256 context
SECURE_STATUS_T sha256_begin(HASH_CONTEXT_T *hctx);                                 //!< SHA-256 context setup
SECURE_STATUS_T sha256_update(HASH_CONTEXT_T *hctx, const void *input, size_t length);   //!< SHA-256 digest update
SECURE_STATUS_T sha256_end(HASH_CONTEXT_T *hctx, uint8_t output[32]);               //!< SHA-256 final digest
SECURE_STATUS_T sha256(const void *input, size_t ilen, uint8_t output[32]);         //!< output = SHA-256(input)
```

Implementing: HMAC-SHA-256

The HMAC construction encapsulates the hash function, and is considered much harder to crack.

Once you have SHA-256, the HMAC-SHA-256 is straightforward to implement.

HMAC has the following construction:

$$\text{HMAC}(K, \text{text}) = \underbrace{H(\underbrace{(K_o \oplus \text{opad}) \,||\, H(\underbrace{(K_i \oplus \text{ipad}) \,||\, \text{text}}_{\text{inner hash}})}_{\text{outer hash}})}$$

Implementing: HMAC-SHA-256 (2)

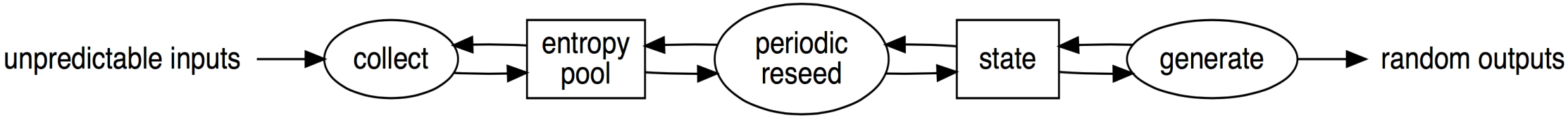

```
SECURE_STATUS_T hmac_sha256_begin(HASH_CONTEXT_T *hctx, const void *key, size_t keylen)
{
    ....
    sha256_begin(hctx);
    return sha256_update(hctx, hctx->ipad, 64);  // H(key xor ipad)
}

SECURE_STATUS_T hmac_sha256_update(HASH_CONTEXT_T *hctx, const void *input, size_t length)
{
    return sha256_update(hctx, input, length);   // hash message blocks
}

SECURE_STATUS_T hmac_sha256_end(HASH_CONTEXT_T *hctx, uint8_t output[32])
{
    sha256_end(hctx, hctx->ipad);                // terminate inner hash: H(Ki // message)
    sha256_begin(hctx);                          // initialize SHA-256 context for outer hash
    sha256_update(hctx, hctx->opad, 64);         // H(Ko)
    sha256_update(hctx, hctx->ipad, 32);         // H(Ko // H(Ki // message))
    sha256_end(hctx, output);                    // output the final hash
    sha256_uninit(hctx);                         // destroy all data in hash context
    return sec_status;
}
```

Random Numbers Generation

Random, unpredictable Numbers are the hardcore of secure systems.

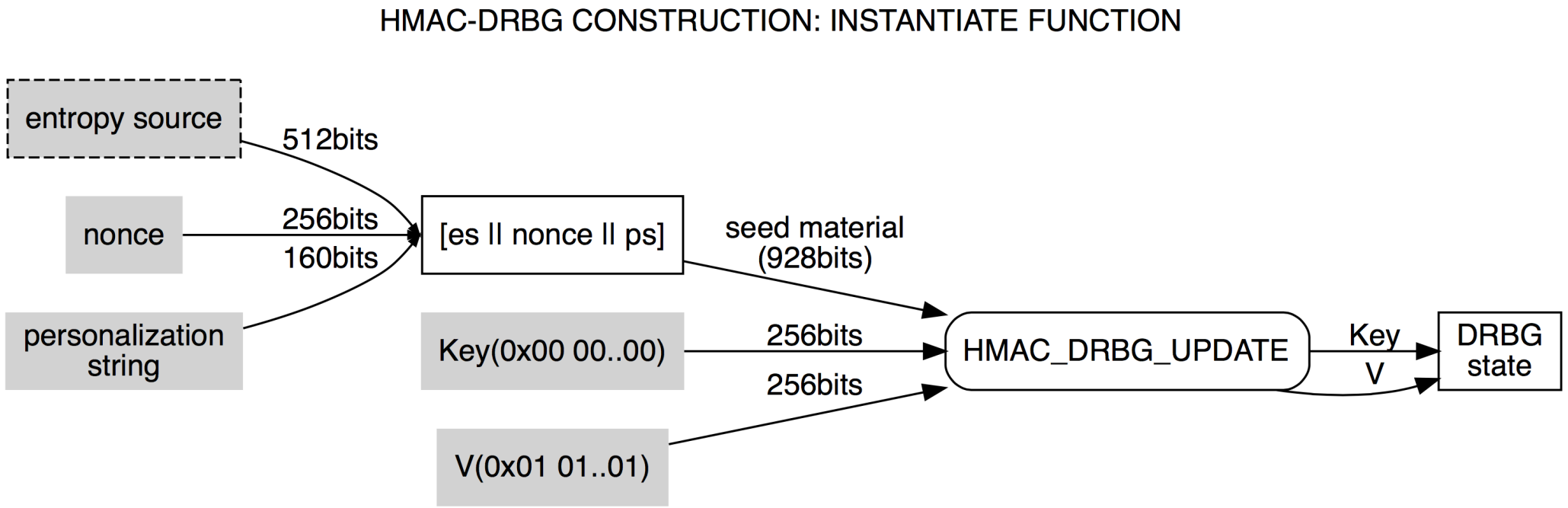
High quality random numbers require a source of unpredictability with a large number of states.

The general structure of a RNG is this:



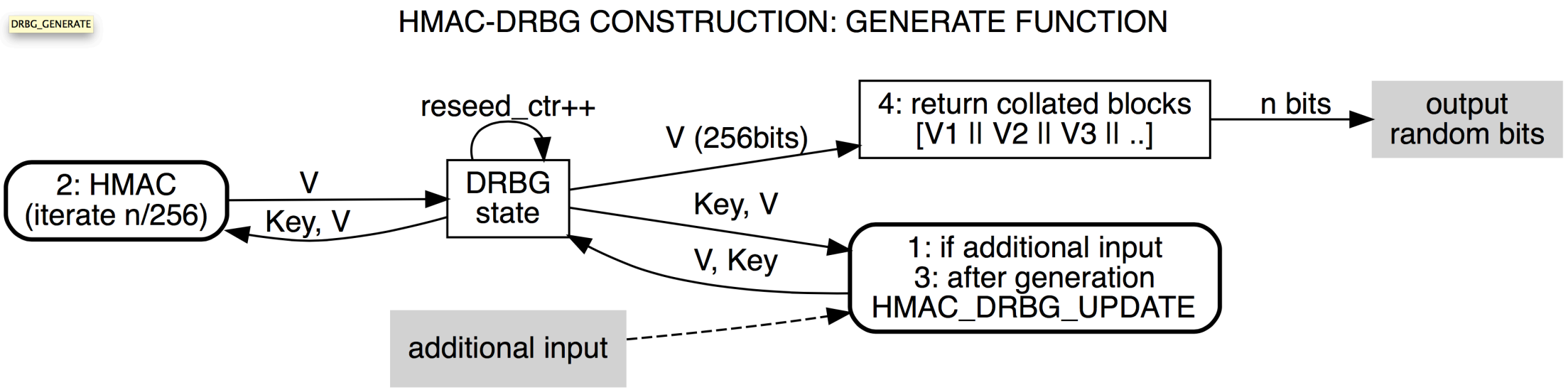
NIST Special PUB 800-90A/B/C define the recommendations for implementing DRBGs.

Implementing: HMAC-DRBG



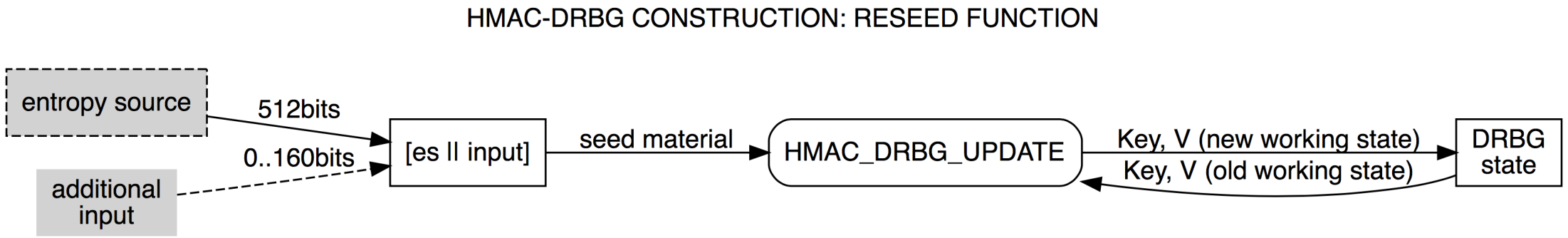
The **`Instantiate`** function creates a new DRBG context.

Implementing: HMAC-DRBG (2)



The **`Generate`** function computes a fresh random output.

Implementing: HMAC-DRBG (3)



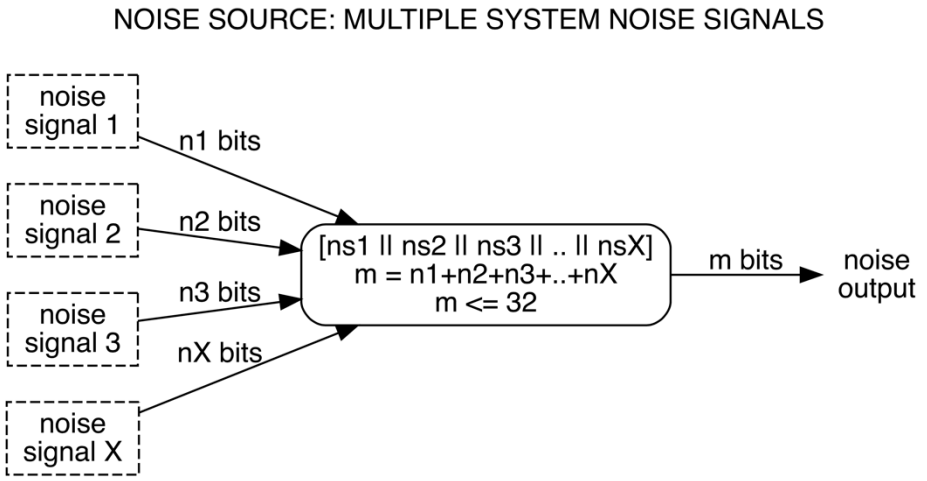
The **Reseed** function restarts the DRBG with fresh entropy data.

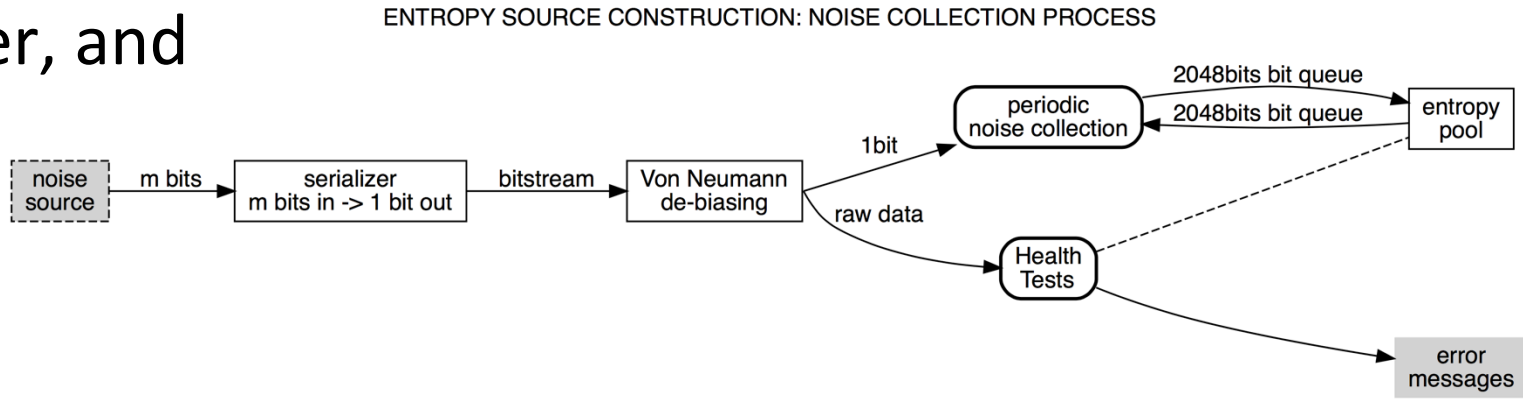
Implementing: Entropy Source

We need uncorrelated noise sources as the source of entropy.

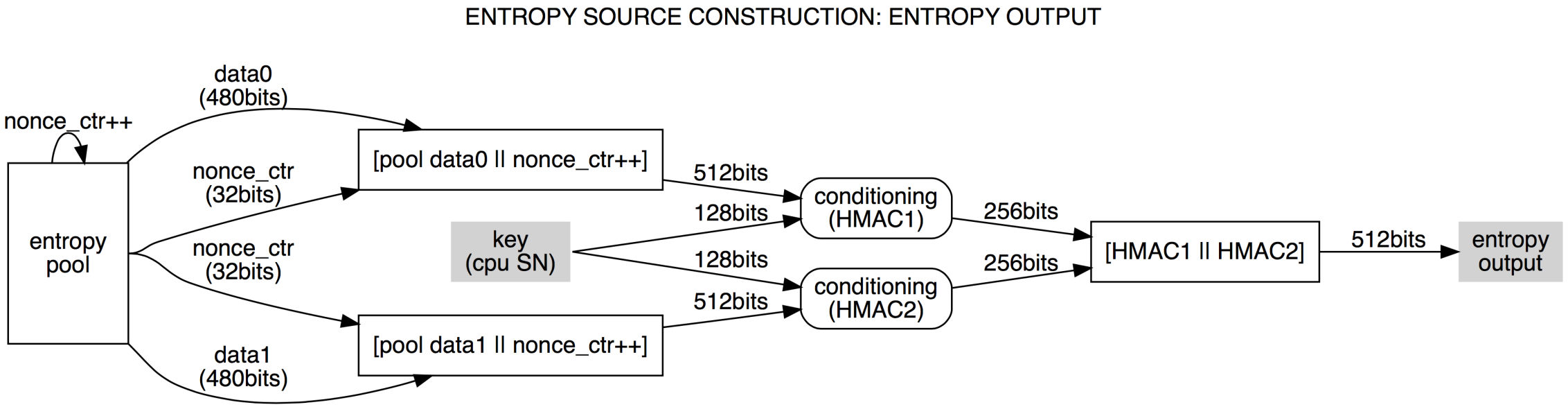
All the ADC analog sensors lower bits are used as noise sources.

The noise is de-biased through a Von Neumann de-bias filter, and collected into a 2048bit entropy pool.





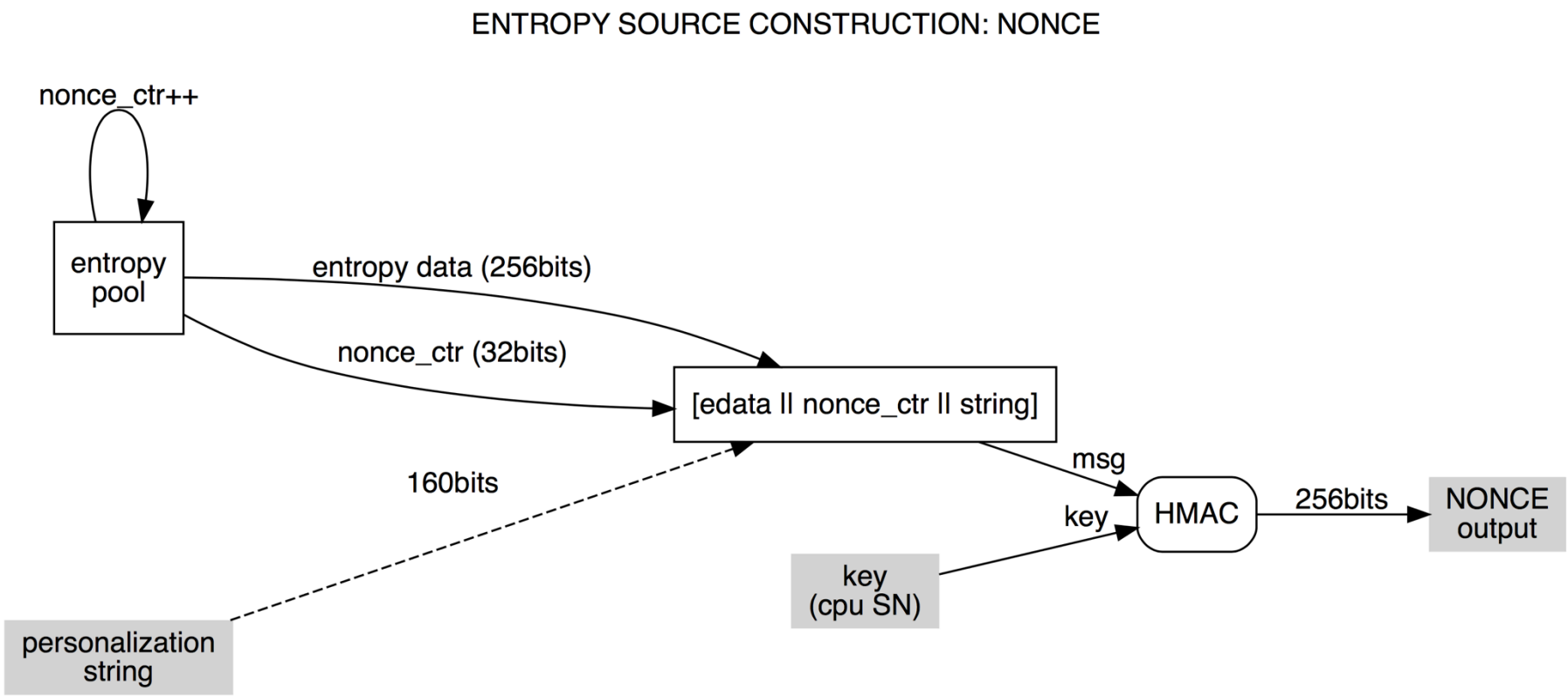
Implementing: Entropy Source (2)



The entropy source takes 960bits of noise data and applies 2 parallel HMAC conditioning functions, using the CPU chip SN as the key.

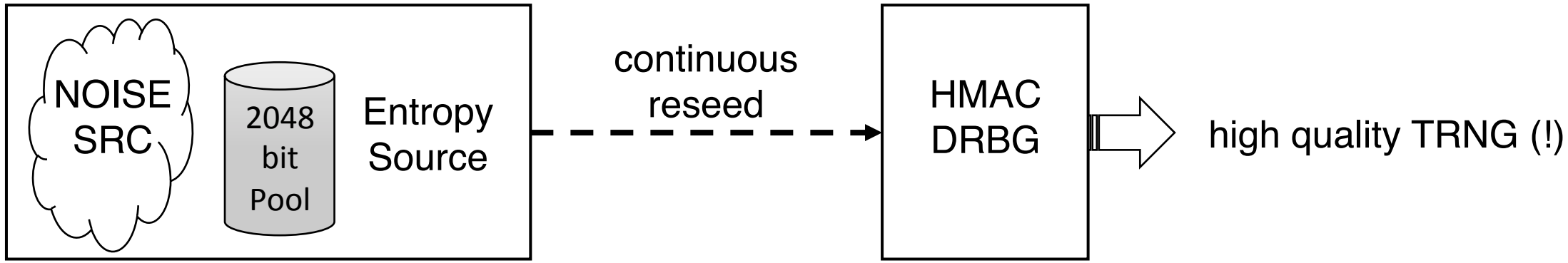
This produces very high quality, very high entropy seed material.

Implementing: Entropy Source (3)



The entropy source also produces high quality cryptographic nonces.

Implementing: (N)RBG



When you add a ‘live’ entropy source to an instantiated HMAC-DRBG, that is continuously reseeded with fresh, high-entropy data, you get a NRBG (Non-Deterministic Random Bit Generator), or an *approved* True Random Number Generator !

This is a fast, high-quality keying material generator for the system.

Digital Signatures: ATECC108A

Elliptic Curves DSA (ECDSA) is one of the strongest public-key cryptographic signature algorithms today.

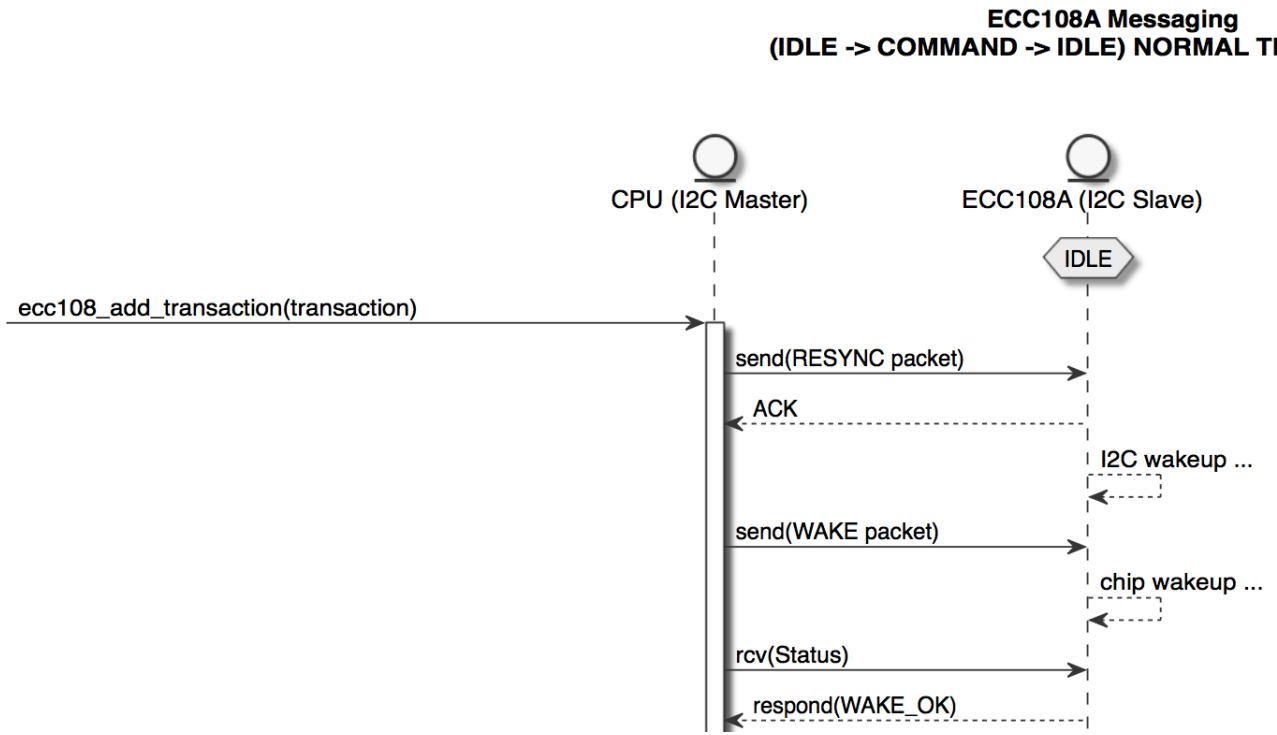
One problem is that it is expensive to implement in a microcontroller.

We selected a certified, external ECDSA accelerator that also works as a secure EEPROM to store the private keys, increasing system compliance to FIPS-140-2 level 2 and 3.

The device is the Atmel (now Microchip) ATECC108A.

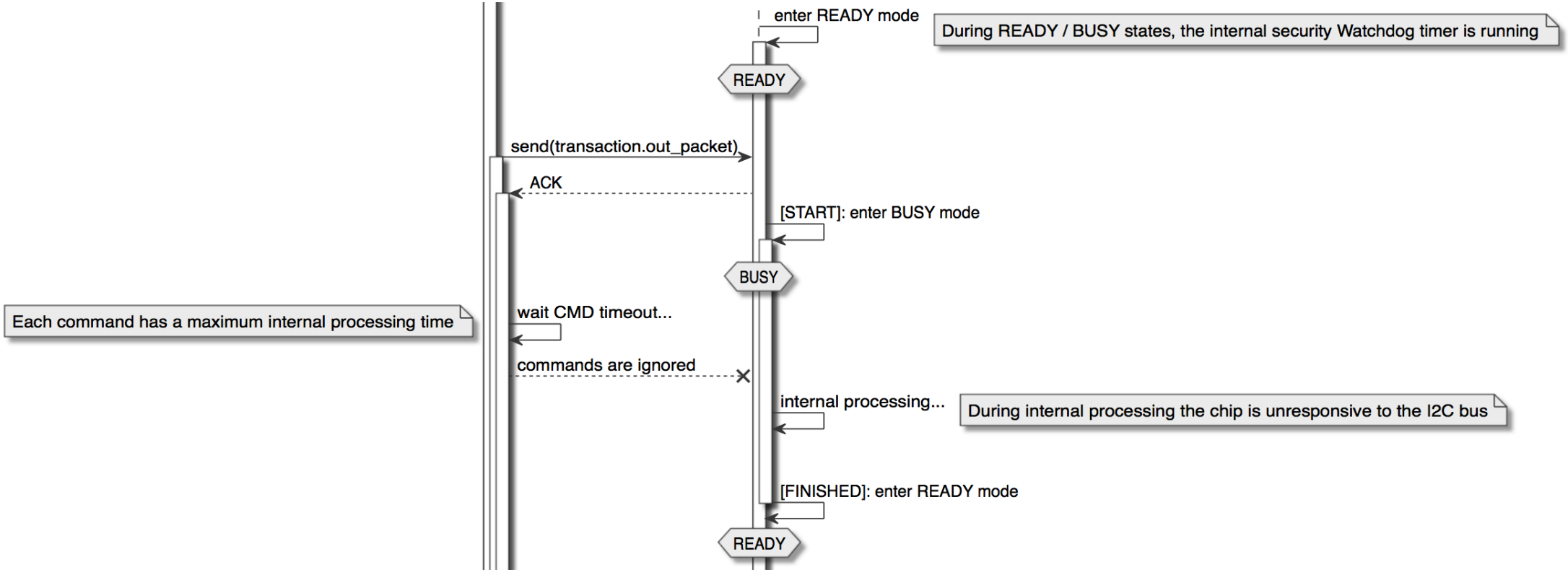
We designed the ECDSA sign/verify as a system service around the crypto memory.

Digital Signatures: ATECC108A (2)



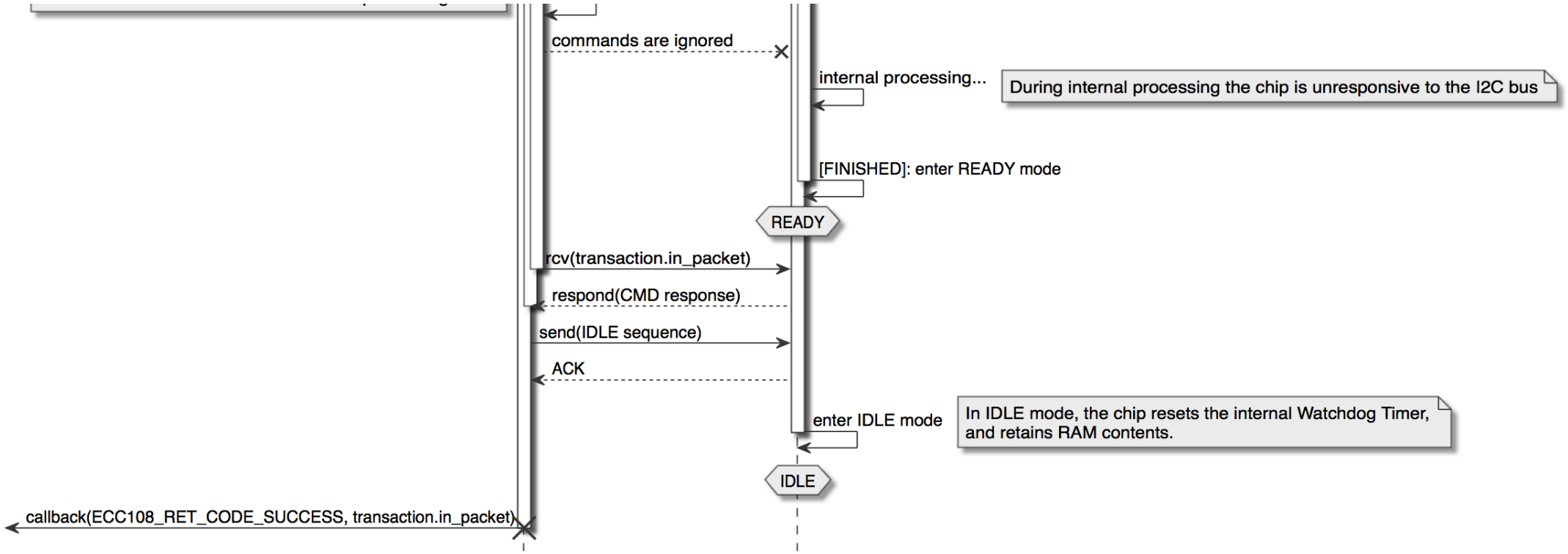
The driver takes the chip out of IDLE state (chip wake up)

Digital Signatures: ATECC108A (3)



The command executes in the BUSY state. The chip is unresponsive during BUSY.

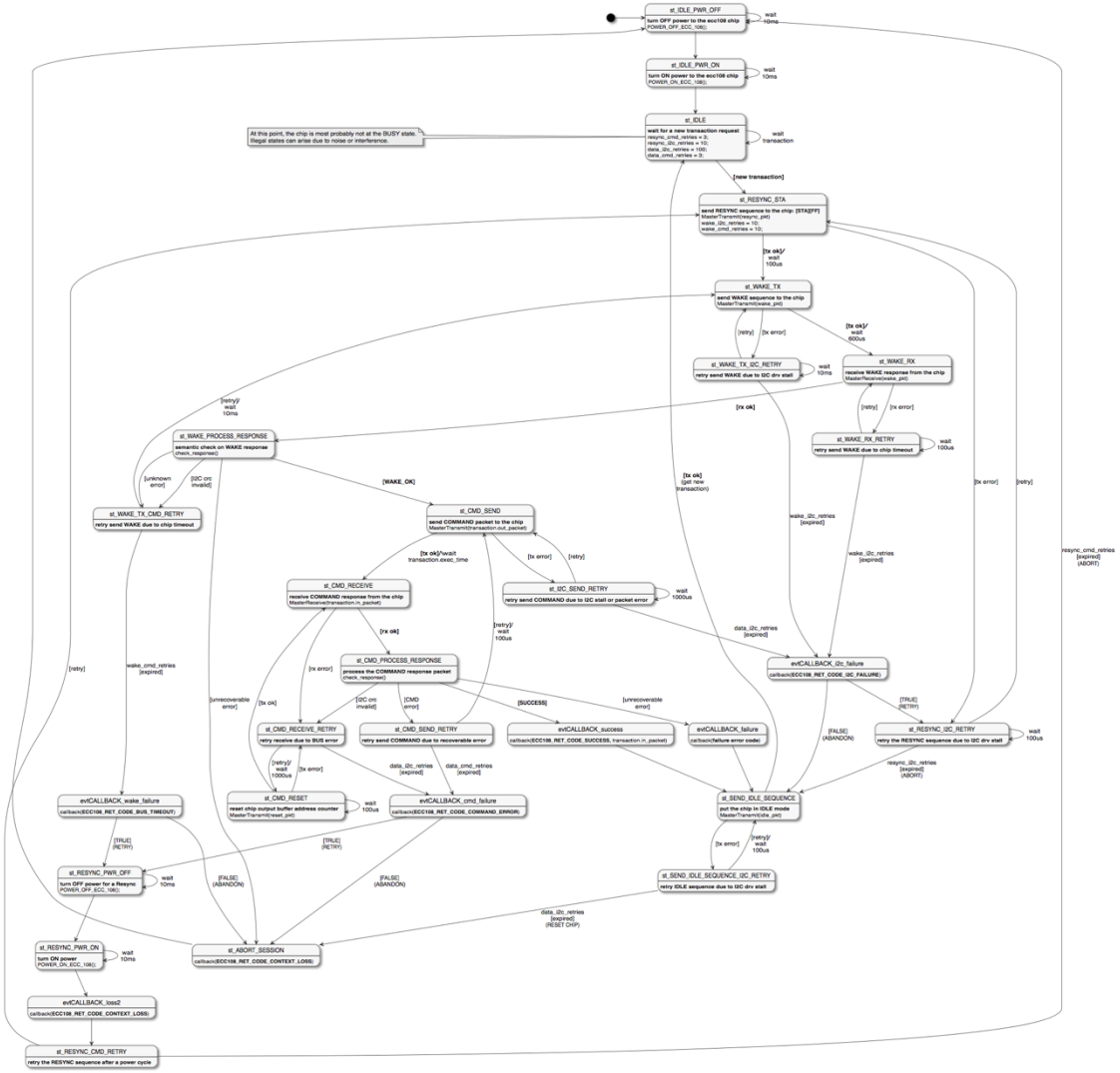
Digital Signatures: ATECC108A (4)



After the command returns, the chip is placed in IDLE again.

Digital Signatures: ATECC108A (5)

The actual driver code accounts for every possible error and bus timeout.



Implementing: ECDSA Sign/Verify

SCML/16
SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

Lib Interface:

```
/** Descriptor for ECC sign/verify operations. */
typedef struct ecdsa_vector_t {
    LAP_TIMER64_T lap_timer64;                  //!< 64 bit lap timer for this operation
    ECC_KEY_TYPES_T key_type;                   //!< Key type, curve and key storage
    uint32_t slot : 8;                          //!< Key slot for internal keys
    uint32_t key_length;                        //!< public key data length
    uint32_t signature_length;                  //!< signature data length
    uint8_t *msg_hash;                          //!< ptr to the 32-byte Message hash (SHA-256)
    uint8_t *key_data;                          //!< ptr to the External Key buffer
    uint8_t *signature;                         //!< ptr to the Signature buffer generated by ecdsa_sign()
} ECDSA_VECTOR_T;

SECURE_STATUS_T ecdsa_sign(ECDSA_VECTOR_T *vector, ECDSA_CMD_EVENT_T *event);
SECURE_STATUS_T ecdsa_verify(ECDSA_VECTOR_T *vector, ECDSA_CMD_EVENT_T *event);
```

RAM secureFence

One of the services available is the secureFence service.

Applications can request that certain structs be locked as *immutable* data.

The structs are signed with a random key and checked every system pass.

If the struct is modified by unauthorized code or wild pointers, the service generates a FAULT, and the system is reset.

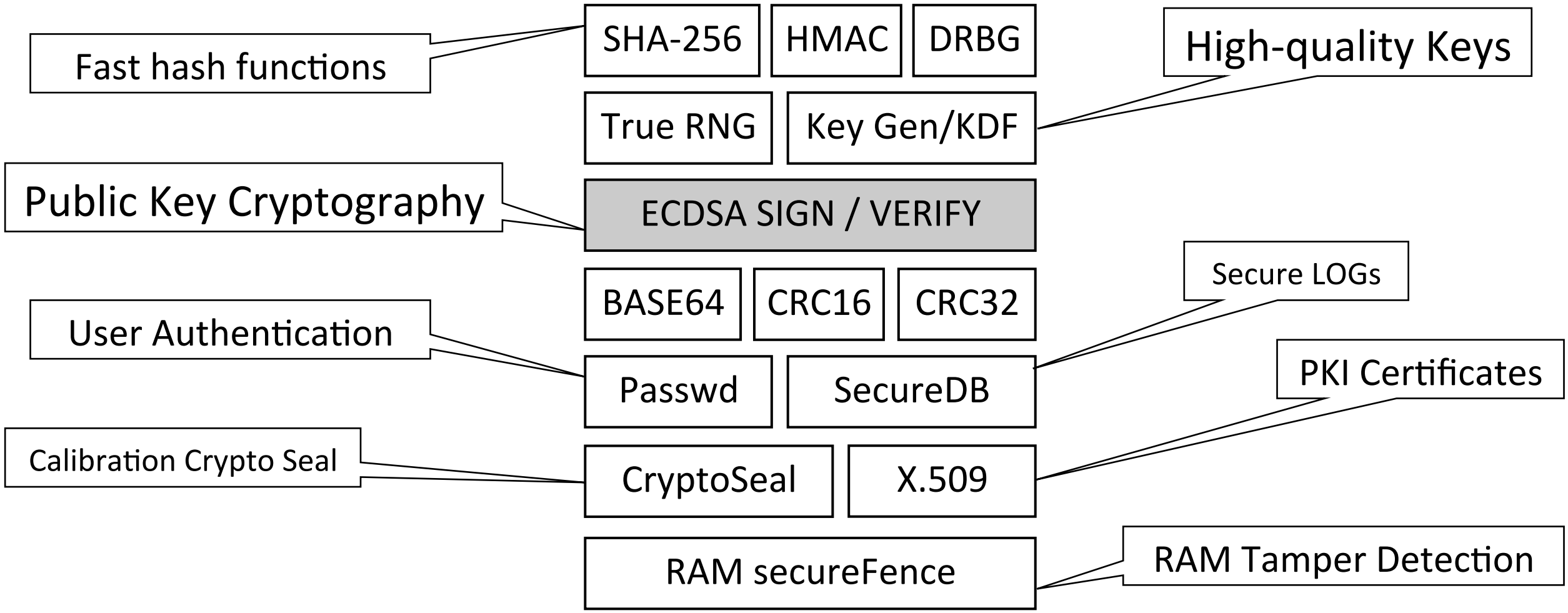
It effectively places a crypto fence around system sensitive data, such as calibration constants and measurement results.

http://i2.wp.com/patriot-tech.com/wp-content/uploads/2014/06/SCADA_security.jpg

RAM secureFence (2)

```
 // --- register secureFence service ---
    if(!register_struct_as_secure(
            &(SECURE_MEM_PARAMETERS_T){
                .struct_ptr         = &weighing_setup,
                .struct_size        = sizeof(weighing_setup),
                .str_struct_name    = "weighing_setup{}"},
            &weighing_data.secure)
    )   throw_app_fault(__FUNCTION__, ": error registering secure struct.");

static WEIGHING_ERROR_T set_adc_sampling_rate(float rate_sps)
{
    WEIGHING_ERROR_T status;

    weighing_data.secure.open();                                    // open vault: allow modifications
    status = adc->set_sampling_rate(rate_sps);
    weighing_setup.sampling_rate = adc->get_sampling_rate();
    status = weighing_data.persistent.save() ? status : ADC_ERR_FAILURE;
    weighing_data.secure.close();                                   // secure vault: protect data

    return status;
}
```

Tamper and Fraud detection



Thank you

Jonny Doin

jonnydoin@gridvortex.com

GridVortex

Thank you

Jonny Doin

jonnydoin@gridvortex.com

SCML/16

SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

SCML/16

SEGURANÇA CIBERNÉTICA EM METROLOGIA LEGAL

Implementing Crypto Security in Bare-Metal Cortex-M

Jonny Doin – GridVortex

#SCML2016