Security and Cryptography 2022

Secure Devices - getting started

Mirosław Kutyłowski

Secure devices is an obligatory and crucial component of most cryptographic systems:

No secure devices $\Rightarrow$ almost the whole crypto is a useless

Example 1: electronic signature

EdDSA signature of a message M with keys x_0, x_1 (irrelevant details omitted):

i. $k := \mathrm{HASH}(M, x_1)$

ii. $r := g^k$

iii. $e := \mathrm{HASH}(M, r)$

iv. $s := k - x_0 \cdot e \bmod q$

v. output (s, e)

What can go wrong:

leakeage of x_1: for a valid signature (s, e) on M

I. recompute k

II. use equality $s = k - x_0 \cdot e$ to derive x_0

Example 2: electronic signature

Schnorr signature of a message M with key x (irrelevant details omitted):

i. k chosen at random

ii. $r := g^k$

iii. $e := \mathrm{HASH}(M, r)$

iv. $s := k - x_0 \cdot e \bmod q$

v. output (s, e)

What can go wrong:

predictable random number generator

I. guess k

II. use equality $s = k - x_0 \cdot e$ to derive x_0

Example 3: DH key exchange

i. Alice chooses a at random, calculates $c_A := g^a$, and sends c_A to Bob

ii. Bob chooses b at random, calculates $c_B := g^b$, and sends c_B to Alice

iii. Alice computes $K := c_B^a$, Bob computes $K := c_A^b$

What can go wrong:

– Bob shares a key ω with Bad Guys

– Bob computes: $b := \mathrm{Hash}(c_A, \omega)$

Bad Guys: observe c_A, compute $b := \mathrm{Hash}(c_A, \omega)$ and $K := c_A^b$

$\Rightarrow$ no session confidentiality at all, Alice has no way to defend

Crypto implementation problems:

A) the original code can be ok, but what about a malware replacing

$$b := \mathrm{PRNG}() \qquad \text{by} \qquad b := \mathrm{Hash}(c_A, \omega)\,?$$

B) what about the manufacturer creating a black box solution with a PRNG with a seed known to the manufacturer?

C) what about mistakes in crypto design or misunderstanding the specification?

Idea:

$\rightarrow$ creating a fully secure operating system, fully secure software etc is practically impossible

→ so let us create a small secure environment interacting with the rest - secure hardware/software device

→ security of such a secure core should suffice

Example implementations:

- SIM cards and smart phones
- DRM tokens
- electronic personal ID card with cryptographic functions
- TPM
- HSM
- ... or even ROM (read only memory) for crucial functions

Recall: Common Criteria

- CC is a general top-down approach with “build from components” strategy
- there is a tradition of bottom-up approaches, trying to cover one-by-one necessary cases, focused on secure cryptographic modules
- the bottom-up approach is more specific to USA, while top-down for Europe

FIPS PUB 140-2, SECURITY REQUIREMENS FOR CRYPTOGRAPHIC MODULES

- Federal Information Processing Standards, NIST, recommendations and standards based on the US law
- for sensitive, but unclassified information (classified information: military, etc)
- levels: 1-4, (just as EAL levels – but the meaning of the levels is different)
- Cryptographic Module Validation Program (certification by NIST and Canadian authority)

Architecture

- **CSP:** Critial security parameters – **all values that must not be leaked** such as cryptographic secret keys, entropy used, PRNG seed and internal state, ...

  focus on protecting them by e.g. separation of logical and physical layer

- **approved security functions:** cryptographic functions to be used inside *e.g. hash functions*
  - compulsory, if to be used in the US public sector,
  - waivers concerning some features are possible (only if the application of the standard makes more harm)
  - other (non-approved) functions may be also implemented

- **CB**: cryptographic boundary, explicit separation of the region where CSP live and are used

  CB need not to take the whole device
- **modes of operation:**
  - secure modes only with approved security functions,
  - insecure mode with not approved functions

Levels (rough description, like for EAL):

- **Level 1**: cryptographic module with at least one approved algorithm, no physical protection (like a PC),

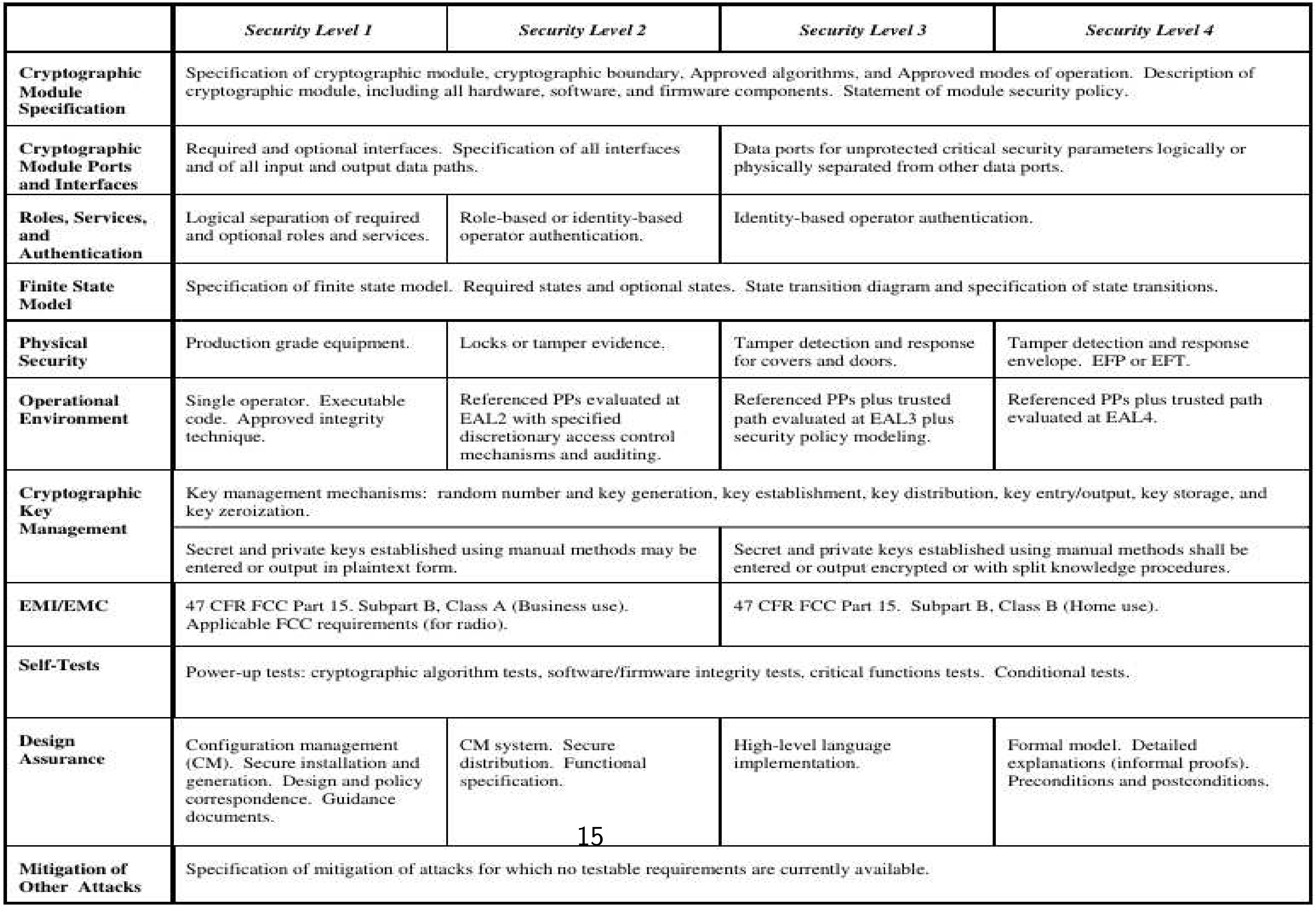
  software implementation of cryptographic functions is possible

- **Level 2**:

  - tamper evident seals for access to CSP (critical security parameters)
  - role base authentication for the operator,
  - refers to certain PPs, EAL2 or higher, or a secure operating system

- **Level 3**:
  - protection against unauthorized access and attempts to modify cryptographic module, detection probability should be high,
  - CSP separated in a physical way from the rest
  - identity based authentication+ role based of an identified person (and not solely role based as on level 2)
  - CSP input and output - encrypted
  - components of the cryptographic module can be executed in a general purpose operating system if
    - PP fulfilled, Trusted Path fulfilled
    - EAL 3 or higher
    - security policy model (ADV.SPM1)
  - or a trusted operating system

- **Level 4**:
  - like level 3 but at least EAL4



	Security Level 1	*Security Level 2*	*Security Level 3*	*Security Level 4*
Cryptographic Module Specification	Specification of cryptographic module, cryptographic boundary, Approved algorithms, and Approved modes of operation. Description of cryptographic module, including all hardware, software, and firmware components. Statement of module security policy.			
Cryptographic Module Ports and Interfaces	Required and optional interfaces. Specification of all interfaces and of all input and output data paths.		Data ports for unprotected critical security parameters logically or physically separated from other data ports.	
Roles, Services, and Authentication	Logical separation of required and optional roles and services.	Role-based or identity-based operator authentication.	Identity-based operator authentication.	
Finite State Model	Specification of finite state model. Required states and optional states. State transition diagram and specification of state transitions.			
Physical Security	Production grade equipment.	Locks or tamper evidence.	Tamper detection and response for covers and doors.	Tamper detection and response envelope. EFP or EFT.
Operational Environment	Single operator. Executable code. Approved integrity technique.	Referenced PPs evaluated at EAL2 with specified discretionary access control mechanisms and auditing.	Referenced PPs plus trusted path evaluated at EAL3 plus security policy modeling.	Referenced PPs plus trusted path evaluated at EAL4.
Cryptographic Key Management	Key management mechanisms: random number and key generation, key establishment, key distribution, key entry/output, key storage, and key zeroization.			
	Secret and private keys established using manual methods may be entered or output in plaintext form.		Secret and private keys established using manual methods shall be entered or output encrypted or with split knowledge procedures.	
EMI/EMC	47 CFR FCC Part 15. Subpart B, Class A (Business use). Applicable FCC requirements (for radio).		47 CFR FCC Part 15. Subpart B, Class B (Home use).	
Self-Tests	Power-up tests: cryptographic algorithm tests, software/firmware integrity tests, critical functions tests. Conditional tests.			
Design Assurance	Configuration management (CM). Secure installation and generation. Design and policy correspondence. Guidance documents.	CM system. Secure distribution. Functional specification.	High-level language implementation.	Formal model. Detailed explanations (informal proofs). Preconditions and postconditions.
Mitigation of Other Attacks	Specification of mitigation of attacks for which no testable requirements are currently available.			

Table 1: *Summary of security requirements*

Documentation

- modes of operation, list of approved and non-approved security functions
- all software and hardware components, CB, and parts not in CB (with explanation why outside CB)
- physical ports and logical interfaces, input and output data paths
- manual or logical controls of a cryptographic module, physical or logical status indicators, and applicable physical, logical, and electrical characteristics
- block diagram of all components with interconnections
- high-level specification languages for software/firmware or schematics for hardware
- specification of all security-related information
- cryptographic module security policy, following the standards

Cryptographic Module Ports and Interfaces

- separate interfaces for Cryptographic Module, but may go through shared physical ports
- 4 obligatory interfaces:
  - data input interface
  - data output interface
  - control input interface
  - status output interface
- *The output data path shall be logically disconnected from the circuitry and processes while performing key generation, manual key entry, or key zeroization.*
- Stronger requirements for Level 3 and 4 (separation)

Roles, Services, and Authentication

- operator - roles - services
- separation of processes if operator can work concurrently with some process
- roles:
  - user
  - crypto officer
  - maintenance
- services: show status to operator, perform self-tests, perform approved security function, bypassing cryptographic operations (bypassing cryptographic processing) if implemented:
  - two independent internal actions shall be required to activate the capability (to prevent activating by mistake)
  - the module shows bypassing status (activated, non-activated, alternating)

Authentication

- pbb of a random guess $<\frac{1}{1000000}$
- one minute attemps: $<\frac{1}{100000}$
- feedback to operator obscured (password character do not appear on the screen, etc)

Finite states model

the same as for smart cards. There is a classification of states helping to separate different activities:

– Power on/off states
– crypto officer states
– key/CSP entry states
– user states
– self-test states
– error states
– bypass states
– maintenence states

Physical security

- full documentation,
- if maintenance functionalities, then many features including erasing the key when accessed
- protecting against probing: one cannot put probing devices through holes
- level 4: environmental failure protection (EFP) features or undergo environmental failure testing (EFT) – prevent leakage through unusual conditions

	General Requirements for all Embodiments	Single-Chip Cryptographic Modules	Multiple-Chip Embedded Cryptographic Modules	Multiple-Chip Standalone Cryptographic Modules
Security Level 1	Production-grade components (with standard passivation).	No additional requirements.	If applicable, production-grade enclosure or removable cover.	Production-grade enclosure.
Security Level 2	Evidence of tampering (e.g., cover, enclosure, or seal).	Opaque tamper-evident coating on chip or enclosure.	Opaque tamper-evident encapsulating material or enclosure with tamper-evident seals or pick-resistant locks for doors or removable covers.	Opaque enclosure with tamper-evident seals or pick-resistant locks for doors or removable covers.
Security Level 3	Automatic zeroization when accessing the maintenance access interface. Tamper response and zeroization circuitry. Protected vents.	Hard opaque tamper-evident coating on chip or strong removal-resistant and penetration resistant enclosure.	Hard opaque potting material encapsulation of multiple chip circuitry embodiment or applicable Multiple-Chip Standalone Security Level 3 requirements.	Hard opaque potting material encapsulation of multiple chip circuitry embodiment or strong enclosure with removal/penetration attempts causing serious damage.
Security Level 4	EFP or EFT for temperature and voltage.	Hard opaque removal-resistant coating on chip.	Tamper detection envelope with tamper response and zeroization circuitry.	Tamper detection/ response envelope with tamper response and zeroization circuitry.

Table 2: *Summary of physical security requirements*

Operational environment:

- **Level 1:** separation of processes, concurrent operators excluded, no interrupting cryptographic module, approved integrity technique
- **Level 2:** operating system control functions under EAL2, specify roles to operate, modify,..., crypto software within cryptographic boundary,

  audit: recording invalid operations, capable of auditing the following events:

  – operations to process audit data from the audit trail,

  – requests to use authentication data management mechanisms,

  – use of a security-relevant crypto officer function,

  – requests to access user authentication data associated with the cryptographic module,

  – use of an authentication mechanism (e.g., login) associated with the cryptographic module,

  – explicit requests to assume a crypto officer role,

  – the allocation of a function to a crypto officer role.
- **Level 3:** EAL3, trusted path (also included in audit trail)
- **Level 4:** EAL4

key management:

- non-approved RNG can be used for IV or as input to approved RNG
- list of approved RNG: refers to an annex and annex to NIST document
- list of approved key establishment methods - again links
- key in/out: automated (encrypted) or manual (splitted in case of Level3 and Level4)

Tests

self-test and power-up. No crypto operation if something wrong. tests based on known outputs

- Pair-wise consistency test (for public and private keys)
- Software/firmware load test
- Manual key entry test
- Continuous random number generator test
- Bypass test – proper switching between bypass and crypto

Design assurance

- configuration management (e.g unique Id numbers)
- delivery assurance
- functional specification
  - L1: naming components responsible for tasks according to the security policy, source code for software, HDL for hardware
  - L2: informal description of cryptographic module, ports, interfaces, and the purpose of the interfaces.
  - L3: high level language description of software and hardware
  - L4: formal description in rigorous notation, proof of model completeness, proof for formal model versus functional specification, source code with comments and preconditions +postconditions

Mitigation of Other Attacks

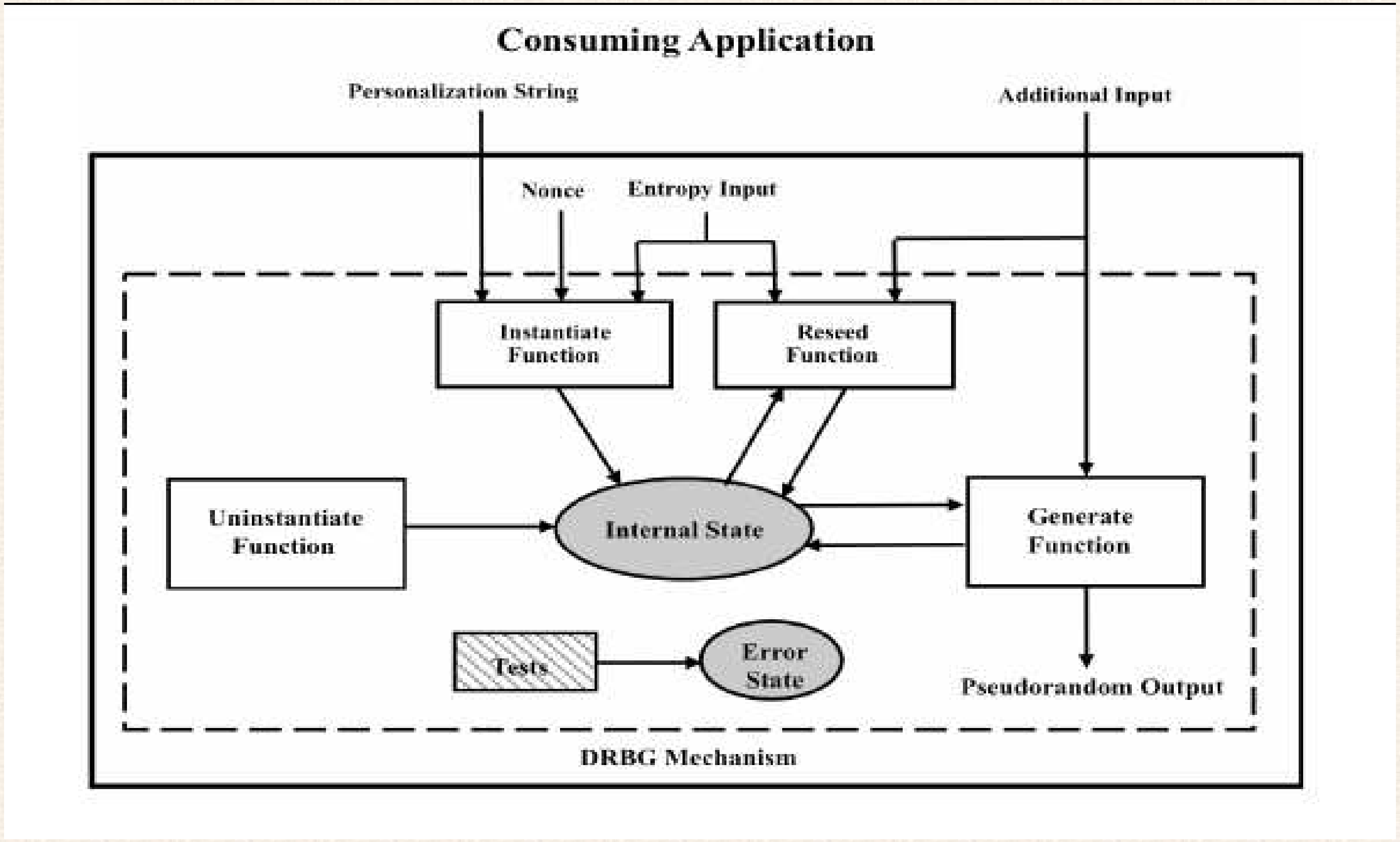
- power analysis: the voltage level depends very much on what operations are executed and on which data
- time analysis: the same, but the computation time may leak the internal state
- fault injection: in case of a fault the original cryptographic security argument does not hold (unless designed ...)
- TEMPEST - electromagnetic leakage

it is known that even the sound emitted by a chip may leak a key (old attack against RSA by A. Shamir)

FIPS Approved Random Number Generators

an example of the NIST approach – standardization of cryptographic functions that are to be deployed on cryptographic secure modules according to FIPS 140-2

- nondeterministic generators not approved,
- deterministic: special NIST Recommendation, in fact “deterministic” means deterministic but with some random input
- first an approved entropy source creates a seed , then deterministic part


Consuming Application
Personalization String
Additional Input
Nonce
Entropy Input
Instantiate Function
Reseed Function
Uninstantiate Function
Internal State
Generate Function
Tests
Error State
Pseudorandom Output
DRBG Mechanism

Instantiation:

- the seed has a limited period, after that period a new seed has to be used
- reseeded function requires a different seed
- different instantiations of a DRNG can exist at the same time, they MUST be independent in terms of the seeds and usage

Internal state:

- it contains cryptographic chain value AND the number of requests so far (each request corresponds to an output)
- different instantiations of DRBG must have separate internal states

Instantiation strength:

- formally defined as "112, 128, 192, 256 bits", intuition: number of bits to be guessed
- `Security_strength_of_output` = min(`output_length`, `DRBG_security_strength`)

Functions executed:

– **instantiate:** initializing the internal state, preparing DRNG to use

– **generate:** generating output bits as DRNG

– **reseed:** combines the internal state with new entropy to change the seed

– **uninstantiate:** erase the internal state

– **test:** internal tests aimed to detect defects of the chip components

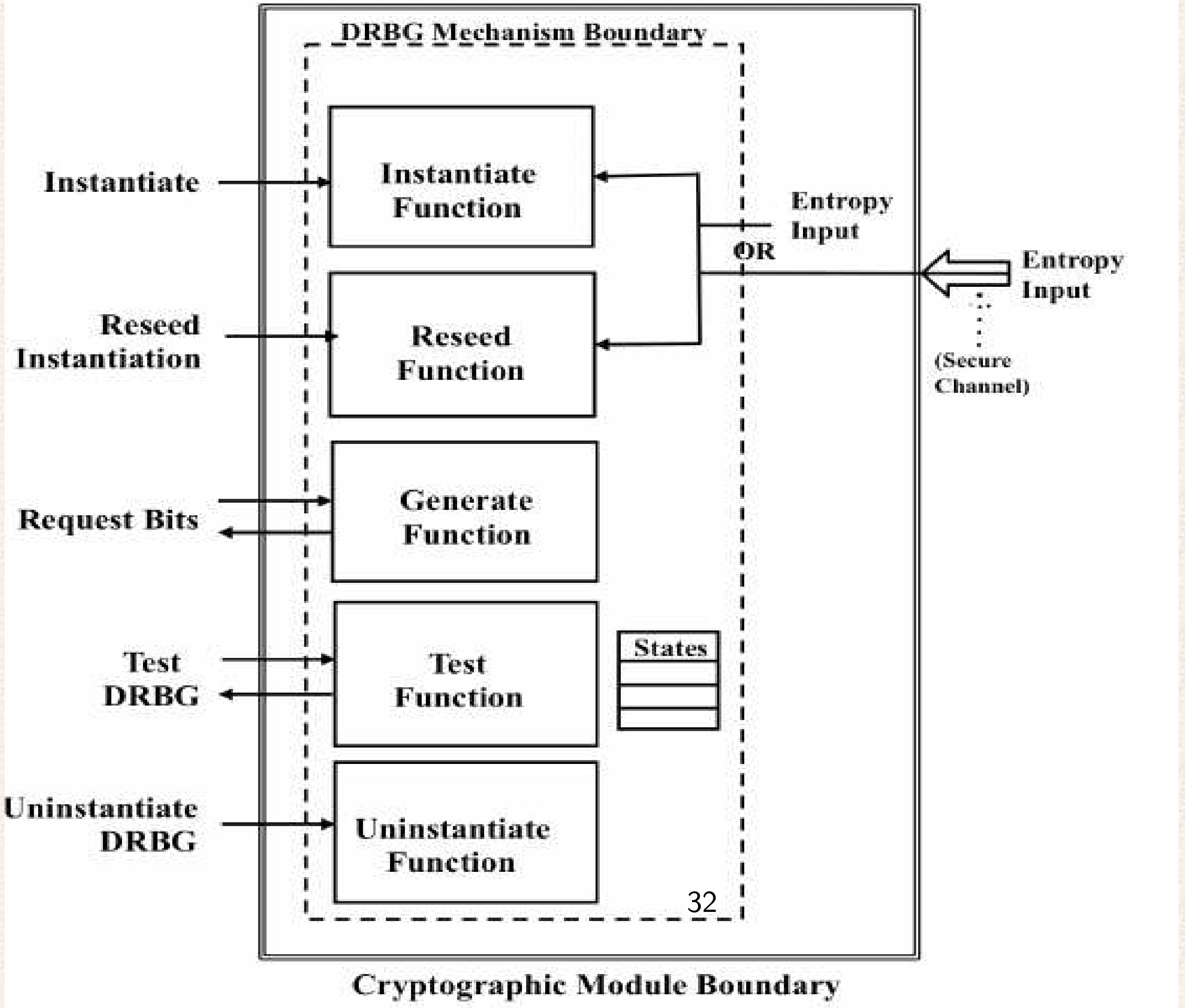
DRBG mechanism boundary:

– this is not a cryptographic module boundary

– DRBG internal state and operation shall only be affected according to the DRBG mechanism specification

– the state exists solely within the DRBG mechanism boundary, it is not accessible from outside

– information about the internal state is possible only via specified output


DRBG Mechanism Boundary
Instantiate
Instantiate Function
Reseed Instantiation
Reseed Function
Request Bits
Generate Function
Test DRBG
Test Function
States
Uninstantiate DRBG
Uninstantiate Function
Entropy Input
OR
Entropy Input
(Secure Channel)
Cryptographic Module Boundary

Seed:

- **entropy** is obligatory, entropy strength should be not smaller than the entropy of the output
- *approved randomness source* is obligatory as an entropy source
- **reseeding**: a nonce is not used, the internal state is used
- **nonce**: it is not a secret. Example nonces:
  - a random value from an approved generator
  - a trusted timestamp of sufficient resolution (never use the same timestamp)
  - monotonically increasing sequence number
  - combination of a timestamp and a monotonically increasing sequence number, such that the sequence number is reset iff the timestamp changes
- not used for any other purposes

reseed operation:

- "for security"
- argument: it might be better than `uninstantiate` and `instantiate` due to aging of the entropy source
- the main difference: the internal state is used! `instantiate` does not use the state

personalization:

- not security critical, but the adversary might be unaware of it (analogous to a login)

resistance:

- **backtracking resistance:** given internal state at time t it is infeasible to distinguish between the output for period $[1, t-1]$ and a random output

- **prediction resistance:** *"Prediction resistance means that a compromise of the DRBG internal state has no effect on the security of future DRBG outputs. That is, an adversary who is given access to all of the output sequence after the compromise cannot distinguish it from random output with less work than is associated with the security strength of the instantiation; if the adversary knows only part of the future output sequence, he cannot predict any bit of that future output sequence that he does not already know (with better than a 50-50 chance)*. – **refers only to reseeding** (before reseeding the output is predictable)

  distinguishability from random input or predicting missing output bits

specific functions:

- (status, entropy_input) = Get_entropy_input (min_entropy, min_ length, max_

  length, prediction_resistance_request),
- **Instantiation:**
  - → check validity of parameters
  - → determine security strength
  - → obtain entropy input and a nonce
  - → run instantiate algorithm to get the initial state
  - → return a handle to this DRNG instantiation

  Instantiate_function(requested_instantiation_security_strength, prediction_resistance_flag, personalization_string)

  prediction_resistance_flag determines whether consuming application may request reseeding

- **Reseed:**
  - → there must be an explicit request by a consuming application,
    - – if prediction resistance is requested
    - – if the upper bound on the number of genereted outpus reached
    - – due to external events

    steps:
  - → check validity of the input parameters,
  - → determine the security strength
  - → obtain entropy input, nonce
  - → run reseed algorithm to get a new initial state
- **Generate function (outputs the bits)**

  Generate_function(state_handle,requested_number_of_bits,requested_security_strength, prediction_resistance_request, additional_input)
- **Removing a DRBG Instantiation**:

  Uninstantiate_function (state_handle)

  internal state zeroized (to prevent problems in case of a device compromise)

Hash_DRBG

comment:

– hash function considered as pseudorandom function

– one can use a random walk: $h_1 = \mathrm{Hash}(V), h_2 = \mathrm{Hash}(h_1), h_3 = \mathrm{Hash}(h_2), \ldots.$

variants:

– hash algorithms: SHA-1 up to SHA-512

– parameters determined, e.g. maximum length of personalization string

– seed length typically 440 (but also 888)

state:

→ value V updated during each call to the DRBG

→ constant C that depends on the seed

→ counter `reseed_counter`: storing the number of requests for pseudorandom bits since new entropy_input was obtained during instantiation or reseeding

instantiation:

```
1. seed_material = entropy_input || nonce || personalization_string
2. seed = Hash_df (seed_material, seedlen)   (hash derivation function)
3. V = seed
4. C = Hash_df ((0x00 || V), seedlen)
5. Return (V, C, reseed_counter)
```

reseed:

```
1. seed_material = 0x01 || V || entropy_input || additional_input
2. seed = Hash_df (seed_material, seedlen)
3. V = seed
4. C = Hash_df ((0x00 || V), seedlen)
5. reseed_counter = 1
6. Return (V, C, reseed_counter)
```

generating bits:

```
1. If reseed_counter > reseed_interval, then return "reseed required"
2. If (additional_input $\neq$ Null), then do
    2.1 w = Hash (0x02 || V || additional_input)
    2.2 V = (V + w) mod $2^{\text{seedlen}}$
3. (returned_bits) = Hashgen (requested_number_of_bits, V)
4. H = Hash (0x03 || V)
5. V = (V + H + C + reseed_counter) mod $2^{\text{seedlen}}$
6. reseed_counter = reseed_counter + 1
7. Return (SUCCESS, returned_bits, V, C, reseed_counter)
```

Hashgen:

```
1. m = $\frac{\text{requested} - \text{no} - \text{of} - \text{bits}}{\text{outlen}}$
2. data = V
3. W = Null string
4. For i = 1 to m
   4.1 w = Hash (data).
   4.2 W = W || w
   4.3 data = (data + 1) mod 2seedlen
5. returned_bits = leftmost (W, requested_no_of_bits)
6. Return (returned_bits)
```

HMAC_DRBG

Update (used for instantiation & reseeding) HMAC_DRBG_Update (provided_data, Key, V):

```
1. Key = HMAC (Key, V || 0x00 || provided_data)
2. V = HMAC (Key, V)
3. If (provided_data = Null), then return Key and V
4. Key = HMAC (Key, V || 0x01 || provided_data)
5. V = HMAC (Key, V)
6. Return (Key, V)
```

Instantiate:

```
1. seed_material = entropy_input || nonce || personalization_string
2. Key = 0x00 00...00
3. V = 0x01 01...01
4. (Key, V) = HMAC_DRBG_Update (seed_material, Key, V)
5. reseed_counter = 1
6. Return (V, Key, reseed_counter)
```

Reseed:

```
1. seed_material = entropy_input || additional_input
2. (Key, V) = HMAC_DRBG_Update (seed_material, Key, V)
3. reseed_counter = 1
4. Return (V, Key, reseed_counter).
```

Generate bits:

```
1. If reseed_counter > reseed_interval, then return "reseed required"
2. If additional_input $\neq$ Null, then
   (Key, V) = HMAC_DRBG_Update (additional_input, Key, V)
3. temp = Null
4. While len (temp) < requested_number_of_bits do:
   4.1 V = HMAC (Key, V)
   4.2 temp = temp || V
5. returned_bits = leftmost (temp, requested_number_of_bits)
6. (Key, V) = HMAC_DRBG_Update (additional_input, Key, V)
7. reseed_counter = reseed_counter + 1
8. Return (SUCCESS, returned_bits, Key, V, reseed_counter).
```

CTR_DRBG

this generator is based on an encryption function, choice: 3DES with 3 keys or AES 128, 192, 256

internal state:

– `V` of `blocklen` bits, updated each time another `blocklen` bits of output are produced

– `Key` of `keylen-bit` bits, updated whenever a predetermined number of output blocks is generated

– `counter (reseed_counter)` = the number of requests for pseudorandom bits since instantiation or reseeding

– `ctr_len` is a parameter depending on implementation, counter field length, at least 4, at most `ctr_len`$\leq$ `blocklen`, for example important when 3DES is used: `ctr_len` is only 64

Update Process: `CTR_DRBG_Update (provided_data, Key, V)`:

where `provided_data` has length `seedlen`

```
1. temp = Null
2.     While (len (temp)$<$ seedlen, do
   2.1 If ctr_len $<$ blocklen /* comment: counter increased in the suffix)
       2.1.1 inc = (rightmost (V, ctr_len) + 1) mod $2^{\text{ctrlen}}$.
       2.1.2 V = leftmost (V, blocklen-ctr_len) || inc
   Else V = (V+1) mod $2^{\text{blocklen}}$
   2.2 output_block = Block_Encrypt (Key, V)
   2.3 temp = temp || output_block
3. temp = leftmost (temp, seedlen)
4. temp = temp $\oplus$ provided_data
5. Key = leftmost (temp, keylen)
6. V = rightmost (temp, blocklen).
```

Instantiate:

1. pad `personalization_string` with zeroes
2. `seed_material = entropy_input` $\oplus$ `personalization_string`
3. `Key` $= 0^{\mathrm{keylen}}$
4. `V` $= 0^{\mathrm{blocklen}}$
5. `(Key, V) = CTR_DRBG_Update (seed_material, Key, V).`
6. `reseed_counter = 1`
7. `Return (V, Key, reseed_counter).`

reseeding is similar

Generate:

```
1. If reseed_counter > reseed_interval, then "reseed required"
2. If (additional_input $\neq$ Null), then
    2.1    temp = len (additional_input).
    2.2    If (temp $<$ seedlem) then  pad additional_input with zeroes
    2.3 (Key, V) = CTR_DRBG_Update (additional_input, Key, V).
    Else additional_input = $0^{seedlen}$
```

```
3. temp = Null
4. While (len (temp)<requested_number_of_bit), do
  4.1     If ctr_len< blocklen
    4.1.1 inc = (rightmost (V, ctr_len) + 1) mod $2^{\text{ctrlen}}$
    4.1.2 V = leftmost (V, blocklen-ctr_len) || inc
  Else V = (V+1) mod $2^{\text{blocklen}}$
  4.2 output_block = Block_Encrypt (Key, V).
  4.3 temp = temp || output_block
5. returned_bits = leftmost (temp, requested_number_of_bits)
6. (Key, V) = CTR_DRBG_Update (additional_input, Key, V)
7. reseed_counter = reseed_counter + 1
8. Return (SUCCESS, returned_bits, Key, V, reseed_counter).
```

Models and solutions based on AES

data:

inside the generator: key, state

from outside: input

leakage:

- only data from computations are leaked
- bounded leakage: λ entropy bits per iteration, some (probabilistic) leakage function
- non-adaptive leakage: leakage function fixed in advance
- simultable leakage: there is always some leakage output. The best situation when the real leakage can be simulated and the adverary cannot distinguish if it is a simulation

knowledge of adversary (some options):

- Chosen-Input Attack (CIA): key hidden, state known, input chosen by adversary
- Chosen-State Attack (CSA): key hidden, state chosen by the adversary, input known
- Known-Key Attack (KKA): key known, state hidden, inputs known

almost not considered: key known, state known, inputs with low entropy and somewhat predictable

Some constructions

Construction 1 (against CIA, CSA,KKA)

- setup: 128-bit string X chosen at random
- initialize: 128 bit strings K and S chosen at random
- generate function (with input I):
  1. $U := K \cdot X^2 + S \cdot X + I$ mod 2^{128}
  2. $S := \mathrm{AES}_U(1)$
  3. output $\mathrm{AES}_U(2)$

Construction 2 (separating entropy extraction and output generation - separating information theoretic arguments from cryptographic procedures)

– setup: 1024-bit strings X, X' chosen at random

– initialize: 1024-bit string S chosen at random

– refresh with I:

  1. $S := S \cdot X + I$

– next (with input I):

  1. $U := [X' \cdot S]_{256}$
  2. $S := \mathrm{AES}_U(1)\mathrm{AES}_U(2)....\mathrm{AES}_U(8)$
  3. $R := \mathrm{AES}_U(9)$