TROPIC01

ODN_TR01_app_002

PIN Verification Application Note

Version: 1.2

Abstract

This application note explains the MAC-and-Destroy PIN verification mechanism and provides implementation guidance for the TROPIC01 secure element developed by Tropic Square.

February 4, 2026

tropicsquare

Contents

Glossary

Abbreviation

- **MCU** – Micro-Controller Unit
- **NVM** – Non-Volatile Memory
- **ROT** – Root-of-Trust
- **OTP** – One Time Programmable memory
- **PUF** – Physically Unclonable Function

Functions and Variables

- *0x9A* – Encoding of one byte in hexadecimal notation (example)
- $A||B$ – Concatenation of two byte arrays A, and B:
  $\{A0, A1\}||\{B0, B1\} \rightarrow \{A0, A1, B0, B1\}$
- *key* – Identifies a value that has a role of a key or a secret.
- *var* – Identifies a value that has a role of a variable. Might be security sensitive, but not critical.

1 Introduction

TROPIC01 is an openly auditable secure element chip that stores cryptographic keys and general-purpose data. It contains protections against multiple types of attacks and is well-suited for Root-of-Trust applications. Among its various capabilities, it supports an operation specifically designed to implement a PIN verification scheme called MAC-and-Destroy.

This application note is written for system architects, embedded and application developers, cryptographers, and security certification authorities.

Overview

MAC-and-Destroy is a novel PIN verification scheme. Its purpose is to release a high-entropy cryptographic key k based on a potentially low-entropy PIN entered by the user. The key k can be then used as a symmetric key to unseal some secret data, as an authentication token, or as a seed input to other cryptography schemes.

The number of invalid PIN entry attempts must be limited to a small number to avoid brute-force attacks. This requires to keep a counter of remaining PIN entry attempts, which is usually vulnerable to fault injection attacks. The MAC-and-Destroy scheme eliminates branching and comparison operations by design, aiming to protect against attacks that reset the PIN attempt counter or disable its decrement.

The scheme also allows for an easy extension to multiple PINs for one high-entropy key, or Wiping PIN extension.

For a threat model of the MAC-and-Destroy PIN verification scheme, see Appendix B.

For a design rationale behind the MAC-and-Destroy PIN verification scheme, see Appendix C.

2 Use Cases and Applications

The MAC-and-Destroy PIN verification scheme is ideal for devices requiring high security and tamper resistance. It is particularly valuable for applications where unauthorized access could have catastrophic consequences, and where devices may face sophisticated physical attacks.

Key applications include:

- Hardware Security Modules (HSMs)
- Access controlled environments
- Temper-resistant hardware wallets for digital assets
- Advanced authentication for TPM-style applications
- Credentials and password managers
- Second layer of two-factor device-to-device or server-to-device authentication
- Authentication tokens

3 Functional Description

The MAC-and-Destroy PIN verification scheme consists of two procedures:

1. **New PIN Setup**
2. **Pin Entry Check**

The **New PIN Setup** procedure takes the user PIN as an input, generates high entropy master secret *s*, initializes the scheme, and returns a 32-byte key *k* as derivative of the *s*.

The **Pin Entry Check** procedure then takes the PIN entered by the user as an input, and checks the PIN. If successful, the correct key *k* is returned.

User may provide additional data alongside the PIN, to link the scheme to a given use-case, device, or as part of a bigger scheme.

3.1 MAC-and-Destroy Slots

The MAC-and-Destroy PIN verification scheme uses slots located in the TROPIC01's flash memory – one slot per PIN entry attempt. These slots are first initialized when a new PIN is being set up. The slots are then invalidated (destroyed) one by one with each incorrect PIN entry attempt. When the correct PIN is entered, the slots are initialized again, therefore the PIN entry limit is reset.

PIN entry attempt fails if:

- PIN is invalid.
- The current slot is not initialized for a given PIN.
- The current slot is destroyed by previous invalid PIN entry attempt.

3.2 MAC-and-Destroy Sequence

The fundamental building block of the MAC-and-Destroy PIN verification scheme is a MAC-and-Destroy (MACANDD) sequence, implemented by TROPIC01:

$$v\_out = \textbf{\textit{MACANDD}}(i, v\_in)$$

The MACANDD sequence

1. Takes a slot index *i* and value *v_in* as an input.
2. Based on the input (*i*, *v_in*), computes value *S_i* and overwrites the slot *i* with it.
3. Outputs value *v_out* as a combination of:
   - the input (*i*, *v_in*)
   - the contents of the slot *i* prior to the execution

The host MCU then implements the full PIN verification scheme using the TROPIC01's MACANDD sequence. For detailed description of the MACANDD sequence, see Appendix A.

3.3 Requirements

MAC-and-Destroy Slots
The scheme requires *n* slots, where *n* is the PIN entry limit. TROPIC01 supports up to 128 slots.

Non-Volatile Memory (NVM)
The scheme also requires the host MCU to be able to store *n* ciphertexts (*c_i*), an authentication tag (*t*), and an information about the number of remaining PIN entry attempts in Non-Volatile Memory. This memory does not require any kind of protection.

Key Derivation Function
The scheme uses a one-way key derivation function (***KDF***), which accepts a 32-byte secret and a diversification string. The function is not expected to support a salt or to have key stretching properties. Any MAC algorithm that returns a 32-byte result is suitable for this purpose. Suitable choices include KMAC256 [5] or HMAC-SHA256 [6].

Symmetric Encryption
Part of the scheme is an encryption (***ENC***) and decryption (***DEC***) of the master secret *s*. This can be any symmetric cipher with 32-byte key, ideally able to encrypt a 32-byte block of data without the need for padding. The security level shall be at least 256-bit, to not weaken the level of the rest of the scheme. Authenticated encryption (e.g. AES-GCM) is not required, but can be used too. Suitable choices are AES-256 [7] or ChaCha20 [8].

Note

If the scheme is implemented exactly as defined in Subsections 3.4 and 3.5, thanks to the way of how this symmetric key is derived, simple bit-wise XOR is also sufficient. However, be careful when modifying the scheme to your needs and use-case. Using standardized symmetric ciphers is always the safe solution.

Additional Data (Optional)
Other confidential data may be added as input to the scheme in order to increase the overall entropy of the input. This should be any data that is difficult to extract by an unauthorized party from the individual hardware components of the device such as component serial numbers or secret values stored in different types of non-volatile memory.

3.4 New PIN Setup

Input: *PIN* chosen by the user, additional confidential data *A*
Output: Cryptographic key *k* (32 bytes)

1. Let *i* = *n*. Store *i* (remaining attempts) to the NVM.
2. Generate a random 32-byte master secret *s*.
3. Compute tag *t* = ***KDF***(*s*, *0x00*), and store it to the NVM.
4. Compute *u* = ***KDF***(*s*, *0x01*).
5. Compute *v* = ***KDF***(*0*, *PIN*||*A*), where *0* is all-zero key.
6. For *i* in *0, ..., n-1*: (each slot)
   - 6.1 Execute ***MACANDD***(*i*, *u*) and ignore the result.
   - 6.2 Execute *w_i* = ***MACANDD***(*i*, *v*).
   - 6.3 Compute *k_i* = ***KDF***(*w_i*, *PIN*||*A*)
   - 6.4 Encrypt *s* → *c_i* = ***ENC***(*k_i*, *s*) and store the resulting ciphertext in the NVM.
   - 6.5 Execute ***MACANDD***(*i*, *u*) and ignore the result.
7. Compute *k* = ***KDF***(*s*, *0x02*).
8. Return *k* and purge all other computed values from RAM.

Note

Optionally, the master secret *s* can be provided as an input to the New PIN Setup procedure.

3.4.1 New PIN Setup – Informal Description

When new PIN is being set up, we first generate master secret s that is used to:

- Derive tag t = ***KDF***(s, *0x00*). (step 3)
- Derive initialization value u = ***KDF***(s, *0x01*). (step 4)
- Derive final key k = ***KDF***(s, *0x02*). (step 7)

We also derive verification value v = ***KDF***(*0*, *PIN*||*A*). (step 5)

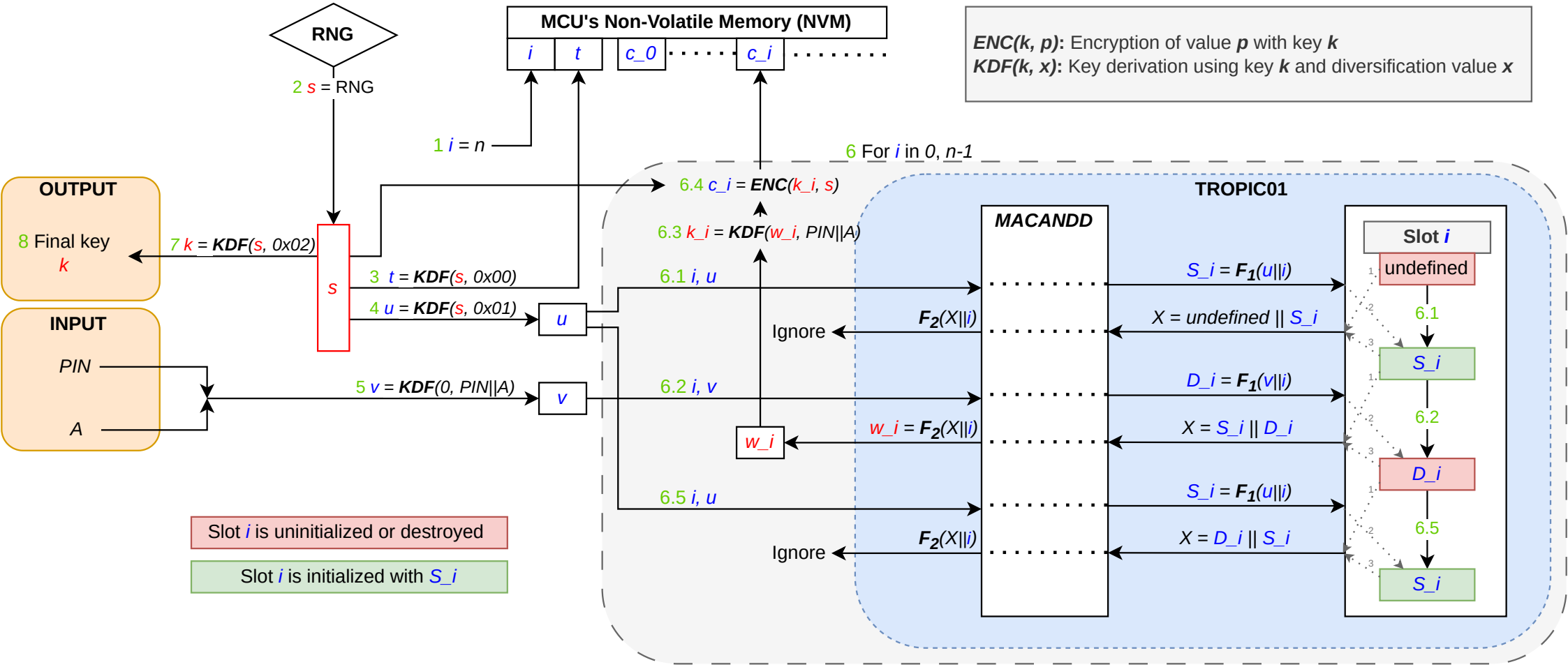
Value u is used to initialize each slot with S_i (step 6.1). Then the initialization value v is passed to the MACANDD sequence to get the key w_i (step 6.2). The value of w depends on:

- The content of the slot $i \Rightarrow S\_i$
- The slot index i
- The verification value $v \rightarrow$ the combination of *PIN* and additional data *A*

From the w_i we then derive an encryption key k_i to encrypt the master secret s (steps 6.3 and 6.4). The resulting ciphertext is then stored in non-volatile memory.

After the step 6.2, the slot i contains $D\_i \neq S\_i$ – is destroyed – therefore it must be initialized again (step 6.5).

Finally, the final key k is derived from the master secret s. (step 7)



New PIN Setup – Flow Diagram

3.5 PIN Entry Check

Input: *_PIN* entered by the user, additional confidential data *A*
Output: Cryptographic key *k* (32 bytes) or FAIL.

1. Load *i* from the NVM.
2. If *i* == 0: FAIL (no attempts remaining)
3. Let *i* = *i* - 1 and store it back to the NVM.
4. Compute *_v* = ***KDF***(*0*, *_PIN*||*A*), where *0* is all-zero key.
5. Execute *_w_i* = ***MACANDD***(*i*, *_v*).
6. Compute *_k_i* = ***KDF***(*_w_i*, *_PIN*||*A*).
7. Read the ciphertext *c_i* from the NVM. Decrypt *c_i* $\rightarrow$ *_s* = ***DEC***(*_k_i*, *c_i*).
8. Compute *_t* = ***KDF***(*_s*, *0x00*).
9. Read the tag *t* from the NVM. If *_t* $\neq$ *t*: FAIL
10. Compute *_u* = ***KDF***(*_s*, *0x01*).
11. For *j* in *i, ... n-1*: (each already destroyed slot)
    11.1 Execute ***MACANDD***(*j*, *_u*) and ignore the result.
12. Let *i* = *n* and store it in the NVM.
13. Compute *k* = ***KDF***(*_s*, *0x02*)
14. Return *k* and purge all other computed values from RAM.

3.5.1 PIN Entry Check – Informal Description

When a PIN entry is being checked, we take the next slot (starting from slot n-1). If there is no slot left ($i = 0$), the maximum amount of attempts was reached (step 2).

If next slot is available, we derive the verification value $\_v = \textbf{\textit{KDF}}(0, PIN||A)$. (step 4) and use it to get the key $\_w\_i$ (step 5). The contents of the slot i is destroyed during this step. Therefore the slot cannot be used again, until its reinitialization.

If:

- The correct PIN and additional data are entered $\Rightarrow$ the derived verification value $\_v$ is correct.
- The slot i is initialized $\Rightarrow$ contains S_i.

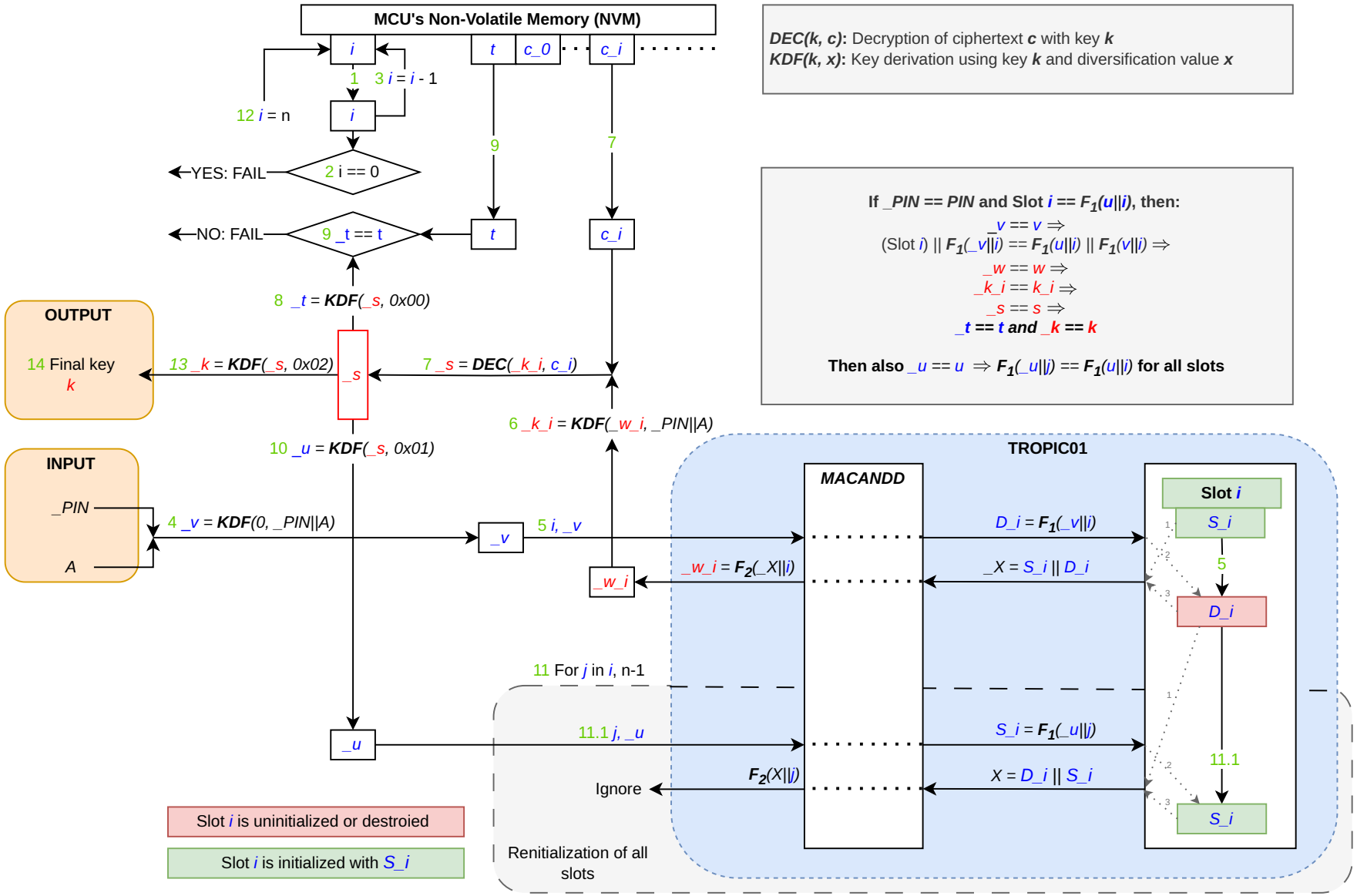
then we get the correct key $\_w\_i = w\_i$. We are then able to correctly derive the decryption key $\_k\_i$ (step 6), decrypt the ciphertext c_i stored during the PIN setup, and obtain the master secret $\_s == s$ (step 7).

If the slot is not initialized, or the entered PIN is incorrect, then $\_w\_i \neq w\_i$, and the decryption of c_i results in incorrect master secret ($\_s \neq s$).

We check if the decrypted master secret $\_s$ is the correct one by computing the tag $\_t$ (step 8) and comparing it to the original tag t (step 9) stored during the PIN setup. If $\_t \neq t$, then $\_s \neq s$, and the PIN check failed.

At the end, the value $\_u$ is derived from the master key $\_s$ (step 10) and the destroyed slots are initialized again (step 11). The final key k is then derived from the master secret $\_s$ (step 14).

Observe that the initialization value u is derived from the master secret s. Therefore without the knowledge of s the slots cannot be reinitialized to the correct value S_i.



PIN Entry Check – Flow Diagram

3.6 Multiple PINs Extension

The MAC-and-Destroy scheme can be extended to support multiple pins. One simply computes multiple

$$v\_j = \textbf{KDF}(0, PIN\_j||A)$$

for each *PIN_j*, and executes step 6 of the New PIN Setup procedure for each *v_j*. Initialization value *u* must stay the the same, as it is derived from the master key *s*. Instead of one ciphertext *c_i* per slot, this results in an array of ciphertexts

$$C\_i = [c1\_i, c2\_i, ..., cj\_i]$$

During the PIN Entry Check procedure, steps 7-9 become a loop through the *C_i* array. If the tag *_t* is equal to the original tag *t* for a ciphertext *cj_i*, for some *j*, then *PIN_j* was entered. If no ciphertext from *C_i* leads to correct tag, invalid PIN was entered.

If it is not desired to be able to tell what PIN was entered, the array *C_i* can be randomly shuffled.

3.7 Wiping PIN Extension

With the use of multiple PINs, the scheme can be further extended to include a Wiping PIN (*WPIN*). The Wiping PIN causes the high-entropy key *k* to be destroyed if the *WPIN* is entered instead of the correct PIN. This extension requires one additional slot.
The idea is to introduce 2 layers of verification:

1. Layer-1 implements Multiple PINs extension with the set of PINs {*PIN*, *WPIN*}, with the use of *n* slots.
2. Layer-2 implements a classic MAC-and-Destroy scheme for the correct *PIN*, with the use of only one slot.

The verification on Layer-1 passes if, and only if, the *PIN* or the *WPIN* was entered. If Layer-1 fails, the verification ends and Layer-2 is not used.

If Layer-1 passes, the verification continues on Layer-2. The Layer-2 then passes if, and only if, the correct *PIN* was entered, and releases the high-entropy key *k*. Because the Layer-2 uses only one slot, if the *WPIN* was entered, the slot is destroyed without releasing the key *k*. Thus it can not be initialized again, and the way for the key *k* is forever closed.

Because the verification enters the Layer-2 if, and only if, Layer-1 passes, it is certain that the PIN checked on Layer-2 is *PIN* or *WPIN*. Therefore the Layer-2 is destroyed only if the *WPIN* is entered.

To mitigate control flow attacks, the array *C_i* of Layer-1 shall be shuffled. Additionally, to link the two layers, the key released by Layer-1 might be used as the additional data for the Layer-2.

See Appendix D for extended New PIN Setup, and PIN Entry Check routines.

4 Usage Examples

This section provides a guide how to use the TROPIC01 SDK (Libtropic [4]) to implement the MAC-and-Destroy PIN verification scheme.

Libtropic contains support only for the MACANDD sequence functionality of TROPIC01. As the overall scheme is implemented by the host MCU, we avoid to provide official support. Instead, Libtropic contains an example implementation. For more information, see the Libtropic documentation [4], specifically the *Tutorials* → *Model* → *Mac-And-Destroy* section, which explains the example.

4.1 How to call the MACANDD sequence

To use the MACANDD sequence implemented by TROPIC01, use the Libtropic's `lt_mac_and_destroy` function:

```
/**
 * @brief Executes the MAC-and-Destroy sequence.
 *
 * @param h           Handle for communication with TROPIC01
 * @param slot        Mac-and-Destroy slot index, valid values are 0-127
 * @param data_out    Data to be sent from host to TROPIC01
 * @param data_in     Data returned from TROPIC01 to host
 *
 * @retval            LT_OK Function executed successfully
 * @retval            other Function did not execute successully, you might use
    lt_ret_verbose() to get verbose encoding
 * of returned value
 */
lt_ret_t lt_mac_and_destroy(lt_handle_t *h, const lt_mac_and_destroy_slot_t slot, const
    uint8_t *data_out, uint8_t *data_in);
```

The function returns a `lt_ret_t` return code. See the possible values in the Libtropic documentation [4]. In general, the function may "fail" in the following cases:

- The `slot` is outside the range of [0, 127].
- The Secure Session is not active or is not authorized to access the given slot.

Use the function as follows. This is a simplified example of the step 6 of the New PIN Setup procedure. We avoid checking of the `retval` for simplicity. We strongly advice users to check the return values of all Libtropic functions.

```c
#include "libtropic.h"

// The initialization value u
uint8_t u[32] = {0};
// The verification value v
uint8_t v[32] = {0};
// Garbage for the ignored results
uint8_t garbage[32] = {0};

// Compute the u and v
...

// Buffer for w used for derivation of the encryption key k_i
uint8_t w[32] = {0};

for (int i = 0; i < n; i++) {
    lt_mac_and_destroy(h, i, u, garbage);   // step 6.1
    lt_mac_and_destroy(h, i, v, w);         // step 6.2
    // Derive k_i from w and encrypt the master secret.
    ...

    lt_mac_and_destroy(h, i, u, garbage);   // step 6.5
}
```

4.2 How to enable or disable slots

Host MCU can limit the access privileges to certain MAC-and-Destroy slots by modifying the CFG_UAP_MAC_AND_DESTROY configuration object. All slots (0-127) are enabled by default. See TROPIC01 Configuration Objects AppNote [3] for more information.

Version history

Version	Date	Description
1.0	30.9.2025	Initial public release
1.1	27.11.2025	Sync all Libtropic parts with release v3.0.0
1.2	30.1.2026	Sync all Libtropic parts with release v3.1.0

References

[1] ODD_TR01_datasheet, Tropic Square

[2] ODU_TR01_user_api, Tropic Square

[3] ODN_TR01_006, Configuration Objects AppNote, Tropic Square

[4] TROPIC01 SDK – Libtropic,
GitHub: `https://github.com/tropicsquare/libtropic`
Documentation: `https://tropicsquare.github.io/libtropic/`

[5] Kelsey, J. , Chang, S. , and Perlner, R. (2016), SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD,
`https://doi.org/10.6028/NIST.SP.800-185`

[6] The Keyed-Hash Message Authentication Code (HMAC), (2008), Federal Information Processing Standards (FIPS), National Institute of Standards and Technology, Gaithersburg, MD,
`https://doi.org/10.6028/NIST.FIPS.198-1`

[7] Advanced Encryption Standard (AES), (2001) Federal Information Processing Standards (FIPS), National Institute of Standards and Technology, Gaithersburg, MD,
`https://doi.org/10.6028/NIST.FIPS.197-upd1`

[8] Yoav Nir and Adam Langley, ChaCha20 and Poly1305 for IETF Protocols, RFC7539
`https://www.rfc-editor.org/info/rfc7539`

[9] Bertoni, G. , Daemen, J. , Debande, N. , Le, T. , Peeters, M. , and Assche, G. V. , “Power Analysis of Hardware Implementations Protected with Secret Sharing”,
`https://eprint.iacr.org/2013/067`

[10] Daemen, J. , “Changing of the Guards: a simple and efficient method for achieving uniformity in threshold sharing”,
https://eprint.iacr.org/2016/1061

[11] Arribas, V. , Nikova, S. , Rikmen, V. , “Guards in Action: First-Order SCA Secure Implementations of Ketje without Additional Randomness”,
https://eprint.iacr.org/2018/806

Legal Notice

Our mission is to provide you with high quality, safe, and transparent products, but to be able to do so, we also have to make the following disclaimers.

To verify the characteristics of our products, consult the repositories we make available on our GitHub. While we do our best to keep the content of these repositories updated, we cannot guarantee that the content of these repositories will always identically correspond to our products. For example, there may be delays in publication or differences in the nature of the software solutions as published and as included in the hardware products. Some parts of our products cannot be published due to third party rights.

We take pride in publishing under open-source license terms, but do not grant licenses to any of our patents. Please consult the license agreement in the repository. We reserve the right to make changes, corrections, enhancements, modifications, and improvements to our products, published solutions and terms at any time without notice.

Since we cannot predict what purposes you may use our products for, we make no warranty, representation, or guarantee, whether implied or explicit, regarding the suitability of our products for any particular purpose.

To the maximum extent permitted by applicable law, we disclaim any liability for any direct, indirect, special, incidental, consequential, punitive, or any other damages and costs including but not limited to loss of profit, revenue, savings, anticipated savings, business opportunity, data, or goodwill regardless of whether such losses are foreseeable or not, incurred by you when using our products. Further, we disclaim any liability arising out of use of our products contrary to their user manual or our terms, their use/implementation in unsuitable environments or ways, or for such use which may infringe third party rights. Notwithstanding the above, the maximum liability from the use of our products shall be limited to the amount paid by you as their purchase price.

A Mac-and-Destroy Sequence in TROPIC01

The MACANDD sequence is used to initialize the slots and then check the PIN entry using given slot. During the check, the slot is destroyed. Two separate KMAC256 [5] cores are used – ***F1*** and ***F2***. Each core has a unique 32-byte key (*KFA*, *KFB*) derived from TROPIC01's internal key source (OTP + PUF). This ensures that intermediate values remain unpredictable and unique to each chip, even when using identical input data from the host MCU.
The MACANDD sequence follows these steps:

1. Initialize ***F2*** with *KFB*.
2. Read value *V1_i* from slot *i*, and use it as the first half of the input to ***F2***.
3. Initialize ***F1*** with *KFA*.
4. Compute *V2_i* = ***F1***(*v_in*||*i*) and store it in slot *i*.
5. Read the value *V2_i* from slot *i* and use it as the second half of the input to ***F2***.
6. Compute *v_out* = ***F2***(*V1_i*||*V2_i*||*i*).

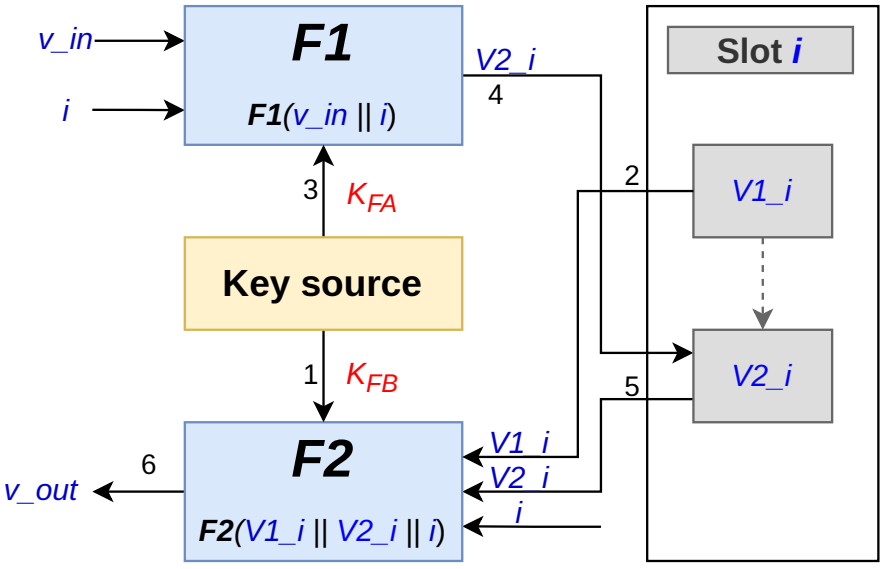


Figure 1: MACANDD sequence diagram

B Threat Model

Reading the Internal Key Source

The TROPIC01 internal key source consists of a physically unclonable function (PUF) and antifuse one-time programmable memory (OTP) and is assumed to be unreadable by the attacker.

Reading the Slots

We assume that reading out data from the slots located in flash memory by an attacker may be possible with physical access to TROPIC01 and non-negligible effort. The ***F2*** computation is therefore keyed with a key from the internal key source so that it is not reproducible by the attacker outside of the TROPIC01.

Writing the Slots

An attacker who would be able to both read from and write arbitrary data to the flash memory would be able to achieve unlimited PIN entry attempts. We assume that writing arbitrary data to the flash by an attacker would require non-negligible effort, which would act as a significant rate limiting countermeasure.
F1 being keyed and preimage resistant function prevents the attacker from easily writing chosen or known value to a slot. This improves resistance to side channel template attacks for an attacker who is not able to read the contents of a slot easily.

Side Channel

The MAC-and-Destroy scheme aims to protect against side channel template attacks by minimizing the control that an attacker may have over the inputs to the MAC functions. Further side channel countermeasures are implemented at the hardware level of the ***F1***, and ***F2*** KMAC256 functions. These countermeasures aim to protect against first-order DPA/CPA-based attacks. They are based on Threshold Implementation presented in [9] with the addition of the "Changing of the Guard" protection described in [10]. These protections have been demonstrated to be effective against first-order attacks in [11].

Fault Injection

The MAC-and-Destroy scheme is designed with the intent to mitigate fault injection attacks by avoiding branching operations and minimizing the attack surface for control flow vulnerabilities, such as skipping an operation.

C Design Rationale

Why the slot is not encrypted?

Slot encryption would introduce greater complexity into the PIN verification scheme which may increase the attack surface for side channel analysis. It may also open greater possibilities for control flow vulnerabilities in the implementation. The MAC-and-Destroy scheme avoids slot encryption by adding the *KFB* key into the ***F2*** computation, so it is not reproducible by the attacker outside of TROPIC01. This way, the scheme is well-suited for a scenario with a potentially readable flash memory by eliminating the complexity of slot encryption.

Why must *V2_i* pass through a slot before being used in *F2*?

This guarantees that the value *V2_i* cannot be processed until the slot contents has been provably overwritten in the flash memory. If the input to ***F2*** would not pass through the slot and the contents of the slot would simply be erased, then an attacker might be able to force the device into skipping over the erasing operation by means of fault injection. Thus the slot would not be depleted and could be used in another PIN entry attempt.

Why is the slot index *i* appended to the input of *F1* and *F2*?

The presence of the slot index in the computation of ***F1*** enforces diversification of the *V1_i* values that are stored in the flash memory. This brings two advantages. Firstly, the diversification improves resilience with respect to side channel template attacks in ***F2***. Secondly, the diversification implies that the index of the slot from which *V1_i* is read influences its value and thus the computation of ***F2***.
The presence of the slot index in the computation of ***F2*** acts as a countermeasure to thwart a particular fault injection attack. An attacker may use fault injection to modify the slot index *i* that is being operated on midway during command execution, so that *V1_i* will be read from slot *i* as it should, but *V2_i* will be written to and then read from a different slot *j*. This way the attacker would avoid depleting slot *i* and achieve possibly unlimited PIN entry attempts. The slot index that is appended to the data input of ***F2*** should specifically be the slot index that *V2_i* was read from. In case of the described attack scenario an incorrect value *j* would be appended, resulting in an invalid output from ***F2***.

Why is the PIN used twice during the PIN check?

It is not possible for a user to externally verify that the TROPIC01 chip is computing what it claims to, because one cannot reproduce the computation without the *KFA*, and *KFB* keys. For example if the TROPIC01 were rigged by the manufacturer to return a low entropy value, such as

$$\textbf{\textit{MAC}}(k\_m, \text{First4Bytes}(\textbf{\textit{Hash}}(V1\_i||V2\_i))),$$

where *k_m* is a key known to the malicious manufacturer, then the user would have no way of finding this out without extensive statistical analysis. This would constitute a backdoor in the scheme. In order to ensure that a rigged TROPIC01 does not weak the security of a user who is using a high-entropy PIN, the PIN needs to be mixed once again with the value returned from the TROPIC01.

The scheme ensures protection of the user even in case the TROPIC01 were to be further rigged to record the command inputs. The verification value *v* is derived from the *PIN* and additional data as

$$v = \textbf{\textit{KDF}}(0, PIN||A),$$

and the output *w_i* from TROPIC01 is then used to derive

$$k\_i = \textbf{\textit{KDF}}(w\_i, PIN||A).$$

This ensures that in the above attack scenario, even when the correct *v* is recorded, the malicious manufacturer is unable to compute *k_i* from the knowledge of *v*, assuming the user's PIN has sufficient entropy.

D Wiping PIN Extension Routines

D.1 New PIN Setup

Input: *PIN* and *WPIN* chosen by the user, additional confidential data *A_L1*
Output: Cryptographic key *k* (32 bytes)

1. Let *i* = *n*. Store *i* (remaining attempts) to the NVM.
2. Generate a random 32-byte secret *s*.
3. Compute tag *t_s* = ***KDF***(*s*, *0x00*) and store it to the NVM.
4. Compute *u_s* = ***KDF***(*s*, *0x01*).
5. Compute *v1* = ***KDF***(*0*, *PIN*||*A_L1*)
6. Compute *v2* = ***KDF***(*0*, *WPIN*||*A_L1*)
7. For *i* in *0, ..., n-1*:
   - 7.1 Execute ***MACANDD***(*i*, *u_s*) and ignore the result.
   - 7.2 Execute *w1_i* = ***MACANDD***(*i*, *v1*).
   - 7.3 Execute ***MACANDD***(*i*, *u_s*) and ignore the result.
   - 7.4 Execute *w2_i* = ***MACANDD***(*i*, *v2*).
   - 7.5 Execute ***MACANDD***(*i*, *u_s*) and ignore the result.
   - 7.6 Compute *k1_i* = ***KDF***(*w1_i*, *PIN*||*A*)
   - 7.7 Encrypt *s* → *c1_i* = ***ENC***(*k1_i*, *s*)
   - 7.8 Compute *k2_i* = ***KDF***(*w2_i*, *WPIN*||*A*)
   - 7.9 Encrypt *s* → *c2_i* = ***ENC***(*k2_i*, *s*)
   - 7.10 Store *C_i* = [*c1_i*, *c2_i*] as a lexicographically sorted list in the NVM.
8. Compute *A_L2* = ***KDF***(*s*, *0x02*)
9. Generate a random 32-byte secret *r*.
10. Compute tag *t_r* = ***KDF***(*r*, *0x00*) and store it in the NVM.
11. Compute *u_r* = ***KDF***(*r*, *0x01*).
12. Compute *v* = ***KDF***(*0*, *PIN*||*A_L1*||*A_L2*).

13. Execute ***MACANDD***(*n*, *u_r*) and ignore the result.
14. Execute *w* = ***MACANDD***(*n*, *v_1*).
15. Execute ***MACANDD***(*n*, *u_r*) and ignore the result.
16. Compute *k_r* = ***KDF***(*w*, *PIN*||*A_L1*||*A_L2*)
17. Encrypt $r \rightarrow c$ = ***ENC***(*k_r*, *r*), and store it to the NVM.
18. Compute *k* = ***KDF***(*r*, *0x02*).
19. Return *k* and purge all other computed values from RAM.

D.2 PIN Entry Check

Input: *_PIN* entered by the user, additional confidential data *A_L1*
Output: Cryptographic key *k* (32 bytes) or FAIL.

1. Load *i* from the NVM.
2. If *i* = 0: FAIL (no attempts remaining)
3. Let *i* = *i* - 1 and store it back to the NVM.
4. Compute *_v* = ***KDF***(0, *_PIN||A_L1*).
5. Execute *_w_i* = ***MACANDD***(*i*, *_v*).
6. Compute *_k_i* = ***KDF***(*_w_i*, *_PIN||A_L1*).
7. Read *C_i* from the NVM.
8. For each *c_i* in *C_i*:
   - 8.1 Decrypt *c_i* $\rightarrow$ *_s* = ***DEC***(*_k_i*, *c_i*)
   - 8.2 Compute *_t_s* = ***KDF***(*_s*, *0x00*).
   - 8.3 If *_t_s* == *t_s*: Compute *u_s* = ***KDF***(*_s*, *0x01*) and break.
9. If *u_s* is None: Fail
10. Compute *A_L2* = ***KDF***(*s*, *0x02*)
11. Compute *_v* = ***KDF***(*0*, *PIN||A_L1||A_L2*).
12. Execute *_w* = ***MACANDD***(*n*, *_v*).
13. Compute *_k_r* = ***KDF***(*_w*, *_PIN||A_L1||A_L2*).
14. Read *c* and tag *t_r* from the NVM.
15. Decrypt *c* $\rightarrow$ *_r* = ***DEC***(*_k_r*, *c*)
16. Compute *_t_r* = ***KDF***(*_r*, *0x00*).
17. If *_t_r* $\neq$ *t_r*: FAIL
18. For *j* in *i, ..., n-1*:
    - 18.1 Execute ***MACANDD***(*j*, *u_s*) and ignore the result.

19. Compute *u_r* = ***KDF***(*_r*, *0x01*).
20. Execute ***MACANDD***(*n*, *u_r*) and ignore the result.
21. Let *i* = *n* and store it in the NVM.
22. Compute *k* = ***KDF***(*_r*, *0x02*)
23. Return *k* and purge all other computed values from RAM.