Identification d'images de micrologiciels

Ambre Iooss
<Prénom>.<Nom>@crans.org

Résumé. Cet article présente deux contributions permettant d'identifier un microcontrôleur à partir d'une image de son micrologiciel. En premier lieu, nous ajoutons à `libmagic` la capacité de reconnaître l'architecture cible d'un micrologiciel. Ensuite, nous proposons une méthode pour identifier les microcontrôleurs ayant les périphériques requis pour l'exécuter. En connaissant les modèles compatibles de microcontrôleurs, il devient alors possible de désassembler le code et d'interpréter les accès mémoire.

1 Introduction : pourquoi identifier des micrologiciels ?

Un microcontrôleur est un circuit intégré contenant un ou plusieurs processeurs, de la mémoire ainsi que des périphériques. Par exemple, une carte de développement Arduino Uno est équipée d'un microcontrôleur Microchip ATmega328P contenant entre autre (voir figure 1 et 2) :

— un processeur d'une architecture AVR 8-bit,
— une mémoire non-volatile de type Flash de 32 kio,
— une mémoire volatile de type SRAM de 2 kio,
— des périphériques USART (port série), SPI et ADC.

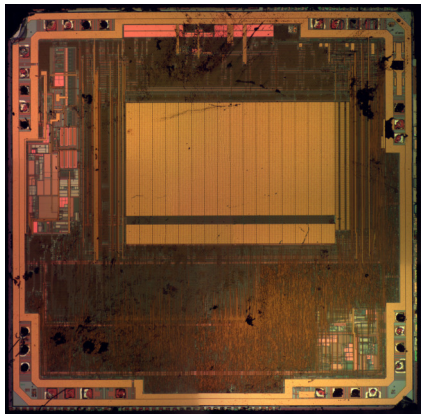

Fig. 1. Microchip ATmega328P, 2990 μm $\times$ 2950 μm, image réalisée par Andrew Zonenberg

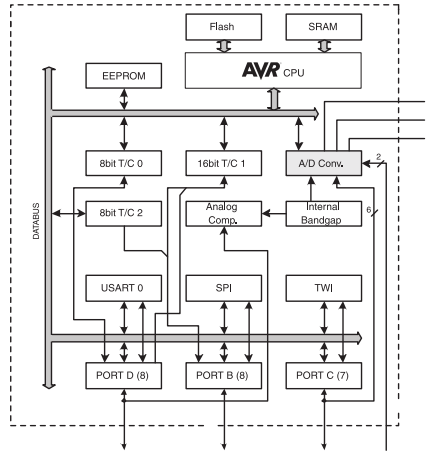

Fig. 2. Extrait du diagramme de la fiche technique de l'ATmega328P

Dans le fonctionnement nominal d'un microcontrôleur, son processeur exécute du code généralement stocké dans une mémoire non-volatile. Ce micrologiciel utilise un sous-ensemble des périphériques du microcontrôleur, par exemple pour envoyer un message sur un port série ou pour contrôler une diode électroluminescente.

Scénario. Il arrive que l'on obtienne l'image binaire d'un micrologiciel sans pour autant connaître le modèle du microcontrôleur qui peut l'exécuter, ni son architecture cible, par exemple en imageant une mémoire d'un système embarqué (voir figure 3), ou en s'intéressant aux binaires distribués dans linux-firmware [1] et dans Linux Vendor Firmware Service (LVFS).

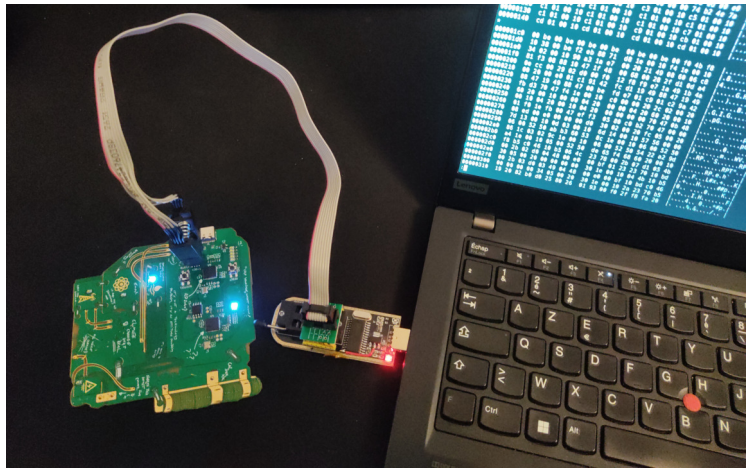

Fig. 3. Lecture avec une pince SPI d'un des micrologiciels de l'épreuve de sécurité matérielle de l'European Cybersecurity Challenge 2022

Connaître le jeu d'instructions facilite l'analyse statique du binaire. Il est aussi souhaitable de connaître les microcontrôleurs destinés à exécuter le code étudié afin de donner du sens aux accès mémoire vers les périphériques, puis potentiellement pour émuler le code.

Des outils statistiques permettent de deviner le jeu d'instructions utilisé [3], mais il est impensable d'ensuite manuellement tenter l'analyse statique pour chaque modèle de microcontrôleur existant : il existe plus de 4 000 modèles de microcontrôleurs exécutant du code ARM Thumb.[2]

[1] Accessible sur `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/`

[2] Nombre de microcontrôleurs ARM Cortex-M disponibles dans ARM Keil début 2025.

Dans quelle mesure l'image d'un micrologiciel est-elle suffisante pour connaître l'architecture du processeur et plus précisément le modèle de microcontrôleurs destiné à l'exécuter ?

Les contributions de cet article répondent à cette problématique, d'abord en améliorant l'outil d'identification binaire `file/libmagic` pour reconnaître l'architecture cible, et ensuite en proposant une nouvelle méthode pour identifier les microcontrôleurs ayant les périphériques nécessaires afin d'exécuter un micrologiciel donné.

2 État de l'art de l'identification binaire

Cette section fait un tour d'horizon des outils d'identification de binaires, puis évalue leur performance sur un ensemble de micrologiciels.

Tableau 1. Outils et bibliothèques d'identification de binaires

Nom	Publication	Licence	Exemples d'utilisation
`file/libmagic` [1]	1987	BSD 2-Clause	Apache HTTP Server, Suricata
`shared-mime-info` [8]	2002	GNU GPL 2	Thunar, Nautilus, Dolphin
`TrID` [7]	2003	Propriétaire	N/A
`binwalk` [4]	2013	MIT	Carving d'une image binaire
`cpu_rec.py` [3]	2017	Apache 2.0	Identification du jeu d'instructions
`PolyFile` [9]	2019	Apache 2.0	Remplacement de libmagic
`binbloom` [5]	2020	Apache 2.0	Identification de l'adresse de base
`Magika` [2]	2024	Apache 2.0	Google Safe Browsing

Le projet `file/libmagic` maintient une large collection de signatures écrites dans un langage dédié afin d'identifier le type d'un fichier binaire. Ce langage permet de décrire un enchaînement de conditions sur le contenu d'un fichier (voir Listing 1). La bibliothèque `shared-mime-info` utilise des conditions plus limitées sur le contenu (e.g. pas de déréférencement d'adresses), mais ajoute des conditions sur le nom du fichier.

`PolyFile` et `binwalk` sont des réécritures en Python du moteur de `file/libmagic`, mais proposent en plus d'identifier récursivement les fichiers incorporés. Ces outils maintiennent leur propre collection de signatures, basées sur celles de `file/libmagic`.[3]

[3] Par exemple binwalk détecte les constantes de SHA2-256 dans le contenu d'un fichier.

Listing 1: Extrait d'une signature pour les fichiers OpenDocument

```
0       string  PK\003\004
>26     string  \x8\0\0\0mimetypeapplication/
>>50    string  vnd.oasis.opendocument.             OpenDocument
>>>73   string  text
>>>>77  byte    !0x2d                               Text
!:mime  application/vnd.oasis.opendocument.text
# [...]
>>>73   string  spreadsheet
>>>>84  byte    !0x2d                               Spreadsheet
!:mime  application/vnd.oasis.opendocument.spreadsheet
```

`TrID` et `Magika` prennent une approche par apprentissage supervisé sur des échantillons pour chaque type de fichier. `TrID` identifie des valeurs fixes au début et à la fin du contenu d'un fichier, alors que `Magika` utilise des techniques d'apprentissage profond nécessitant plus d'échantillons.

`cpu_rec` utilise également une technique d'apprentissage, mais uniquement dans le but d'identifier l'architecture du processeur cible. Néanmoins, parce que certains jeux d'instructions sont des sous-ensembles d'autres jeux d'instructions, cette approche n'indique pas précisément la variante utilisée. Par exemple, `cpu_rec` indique `ARMhf` (ARMv7 hard-float) dans le cas d'un binaire ne contenant que des instructions compressées ARM Thumb (voir Listing 2). De plus, l'information sur le jeu d'instruction n'est pas suffisante pour connaître les modèles de microcontrôleurs susceptibles d'exécuter ce code.

Listing 2: Analyse cpu_rec d'un micrologiciel ARM Cortex-M

```
~ ./cpu_rec.py AM32_F415_BOOTLOADER_PA0_CAN_V13.bin
AM32_F415_BOOTLOADER_PA0_CAN_V13.bin    full(0x38e4)   ARMhf
↪  chunk(0x3400;26)    ARMhf
```

`binbloom` permet de trouver l'endianness et l'adresse de chargement d'une image de micrologiciel binaire avec un arbre de décision utilisant les pointeurs comme point d'intérêt.

Performance dans le cas de micrologiciels. Nous prenons quelques échantillons disponibles sous licence libre des projets AM32-MultiRotor-ESC-firmware (GPL 2), Betaflight (GPL 3), freemodbus (BSD), MarlinFirmware (GPL 3), MicroPython (MIT), QMK (GPL 2) et Tock (MIT). L'analyse de ces échantillons par les différents outils (tableau 2) montre que la majorité des outils d'identification actuels ne peuvent pas identifier de façon fiable ces fichiers.

Tableau 2. Analyse d'échantillons avec les différents outils

	file 5.39	shared-mime-info 2.4	TrID 2.24	binwalk 2.3.4
AM32_G071_BOOTLOADER_PA2_V13.bin	data	Inconnu	Inconnu	Rien
AM32_V203_BOOTLOADER_PA6_V13.bin	data	Inconnu	Inconnu	Rien
betaflight_4.4.2_STM32H750.hex.bin	data	Inconnu	Inconnu	Rien
freemodbus_MCF5235.bin	data	Inconnu	Inconnu	Rien
marlin_STM32F103RE_btt.bin	data	Inconnu	Inconnu	Rien
marlin_teensy41.hex.bin	data	Inconnu	Inconnu	Rien
micropython_RPI_PICO-20240602-v1.23.0.uf2	data	Inconnu	Inconnu	AES, SHA256
micropython_STM32F411DISC-20240602-v1.23.0.dfu	data	Inconnu	Inconnu	SHA256
micropython_PCA10059-20241129-v1.24.1.bin	data	Inconnu	Inconnu	SHA256
qmk_avr_ydkb_just60_default.bin	data	Inconnu	Inconnu	Rien
tock_imxrt1050-evkb.bin	data	Inconnu	Inconnu	Rien
tock_nano_rp2040_connect.bin	data	Inconnu	Inconnu	Rien

	cpu_rec.py 1.1	PolyFile 0.5.1	binbloom 2.1	Magika 0.5.1
AM32_G071_BOOTLOADER_PA2_V13.bin	ARMhf	data	LE, 0x22802000	ISO 9660 CD-ROM
AM32_V203_BOOTLOADER_PA6_V13.bin	RISC-V	data	LE, 0x00000000	pcap capture
betaflight_4.4.2_STM32H750.hex.bin	ARMhf	bitmap	LE, 0x24010000	Inconnu
freemodbus_MCF5235.bin	M68k	data	LE, 0x5e4e1000	Inconnu
marlin_STM32F103RE_btt.bin	ARMhf	data	LE, 0x08007000	Inconnu
marlin_teensy41.hex.bin	ARMhf	data	LE, 0x1ffec000	Inconnu
micropython_RPI_PICO-20240602-v1.23.0.uf2	IA-64	UF2	LE, 0x0ffc6000	Inconnu
micropython_STM32F411DISC-20240602-v1.23.0.dfu	ARMhf	data	BE, 0x08130000	Inconnu
micropython_PCA10059-20241129-v1.24.1.bin	ARMhf	bitmap	LE, 0x00001000	MP4 media
qmk_avr_ydkb_just60_default.bin	AVR	data	BE, 0xcdb7d000	Inconnu
tock_imxrt1050-evkb.bin	ARMhf	data	LE, 0x60000000	ISO 9660 CD-ROM
tock_nano_rp2040_connect.bin	ARMhf	data	LE, 0x10000000	ISO 9660 CD-ROM

3 Identification de l'architecture du microcontrôleur

Plutôt que d'utiliser une approche statistique comme `cpu_rec.py` pour identifier l'architecture du processeur cible, cette section présente une approche qui se base sur des caractéristiques observées dans les tables de vecteurs d'interruptions. Nous détaillons l'implémentation de cette approche pour les architectures ARM Cortex-M et AVR.

Table de vecteurs d'interruptions. De nombreuses architectures de microcontrôleurs utilisent une table de vecteurs d'interruptions afin de pointer le code à exécuter après une interruption. Par exemple, le vecteur `HardFault` en ARM Cortex-M pointe une fonction à exécuter lorsque le processeur exécute du code erroné. Cette table est généralement placée à une adresse fixe vers le début de la mémoire et contient un vecteur de réinitialisation pointant sur le point d'entrée du code (exemples dans les Listings 3 et 4).

L'idée ici est d'utiliser des spécificités des tables de vecteurs de chaque architecture pour l'identification. Pour éviter d'écrire des conditions trop génériques qui mèneraient à des faux positifs, il est souhaitable d'effectuer une correspondance sur au moins 16 bits du binaire, préférablement 32 bits uniques.

Listing 3: Vecteurs Cortex-M

```
00  00 00 04 20   Pointeur de pile
04  c1 d0 02 00   Reset
08  31 9c 02 00   NMI
0C  7b f1 01 00   HardFault
10  31 9c 02 00   MemManage
14  31 9c 02 00   BusFault
18  31 9c 02 00   UsageFault
1C  00 00 00 00   (Réservé)
20  00 00 00 00   (Réservé)
24  00 00 00 00   (Réservé)
28  00 00 00 00   (Réservé)
2C  31 9c 02 00   SVCall
    [...]
```

Listing 4: Vecteurs AVR

```
00  0c 94 74 01   RESET
04  0c 94 ac 01   INT0
08  0c 94 ac 01   INT1
0C  0c 94 ac 01   INT2
10  0c 94 ac 01   INT3
14  0c 94 ac 01   INT4
18  0c 94 ac 01   INT5
1C  0c 94 ac 01   INT6
20  0c 94 ac 01   INT7
24  0c 94 ac 01   PCINT0
28  0c 94 73 18   PCINT1
2C  0c 94 44 19   PCINT2
    [...]
```

3.1 Identification de l'architecture ARM Cortex-M

ARM spécifie le début des tables de vecteurs pour ses cœurs Cortex-M, et laisse l'intégrateur ajouter des vecteurs additionnels à la fin. L'agencement de la mémoire est aussi standardisé pour ARM Cortex-M [6] :

— `0x00000000 - 0x1FFFFFFF` : mémoire non-volatile (code),
— `0x20000000 - 0x3FFFFFFF` : mémoire volatile (SRAM),

— `0x40000000 - 0x5FFFFFFF` : périphériques du circuit intégré,
— `0x60000000 - 0x9FFFFFFF` : RAM externe,
— `0xA0000000 - 0xDFFFFFFF` : périphériques externes,
— `0xE0000000 - 0xE00FFFFF` : périphériques internes au cœur.

Les tables de vecteurs ARM Cortex-M commencent par le pointeur initial de pile, qui est chargé dans le registre `sp` (*stack pointer*) lors de la réinitialisation. Ce pointeur vise le haut de la pile en SRAM.[4] Les cœurs ARM Cortex-M sont little-endian et ont généralement moins de 16 Gio de SRAM.[5] Donc le premier octet de la table de vecteurs est `0x20`.

Les cœurs ARM Cortex-M exécutent du code compressé ARM Thumb, donc avec des pointeurs d'adresses impaires.[6] Les vecteurs pointent vers des fonctions qui sont généralement stockées en mémoire non-volatile. Nous pouvons donc vérifier que les vecteurs de la table d'interruptions sont bien impairs et situés avant `0x20000000`. Enfin, certains vecteurs sont réservés. La majorité des environnements de développement mettent ces vecteurs à `0x00000000` ou `0xFFFFFFFF`. Grâce à ces critères, nous pouvons écrire une signature (voir Listing 5).

Listing 5: Signature ARM Cortex-M pour `file/libmagic`

```
# Octet de poids fort de la pile
3              byte                    0x20
# Les pointeurs sont impairs et avant 0x20000000
>4             ulelong&0xE0000001      0x00000001
>>8            ulelong&0xE0000001      0x00000001
>>>12          ulelong&0xE0000001      0x00000001
>>>>44         ulelong&0xE0000001      0x00000001
>>>>>56        ulelong&0xE0000001      0x00000001
# Sections réservées Cortex-M
>>>>>>28       ulelong+1               <2
>>>>>>>32      ulelong+1               <2
>>>>>>>>36     ulelong+1               <2
>>>>>>>>>40    ulelong+1               <2
>>>>>>>>>>52   ulelong+1               <2          ARM Cortex-M
```

Évaluation de la signature. Cette signature a été testée avec des échantillons de micrologiciels provenant de ces projets libres : AM32 Bootloader, Betaflight, Marlin, IronOS, QMK, RMK, Tock et MicroPython. Cette base d'échantillons a permis d'éclaircir certaines exceptions. Par exemple les microcontrôleurs NXP FlexSPI et Raspberry Pi RP2040 ajoutent du

[4] Sa valeur donne une indication sur la taille totale de SRAM du microcontrôleur.
[5] À cause de limitations de densité, la DRAM est plus adaptée pour de telles capacitées.
[6] La parité du pointeur active ou désactive le mode compressé lors d'un saut.

code en amont de la table de vecteur. Ces exceptions peuvent être gérées avec des variantes de la signature précédente (voir Listing 6).

Listing 6: Signature Raspberry Pi RP2040 pour `file/libmagic`

```
# Bootloader stage 2, sources dans le pico-sdk
# boot2_*.S code (_stage2_boot)
0              ulelong                 0x4B32B500
>4             ulelong                 0x60582021
>>8            ulelong                 0x21026898
# exit_from_boot2.S code (check_return) `pop {r0}; cmp r0, #0`
>>>148         ulelong                 0x2800bc01
# continue ensuite avec la signature ARM Cortex-M précédente.
```

3.2 Identification de l'architecture Microchip AVR

AVR utilise une table de vecteurs sous forme d'instructions de saut. Il est donc possible d'utiliser l'opcode de ces instructions pour identifier le type du fichier binaire. Les cœurs AVR utilisent soit des instructions `JMP` sur 4 octets, ou des instructions `RJMP` sur 2 octets (pour les cœurs AVR plus petits).

Listing 7: Signature Microchip AVR pour `file/libmagic`

```
# JMP (4 octets) pour Reset, Int0-2, PcInt0-3 and WDT
0           uleshort&0xFE0E     0x940C
>4          uleshort&0xFE0E     0x940C
>>8         uleshort&0XFE0E     0x940C
>>>12       uleshort&0XFE0E     0x940C
>>>>16      uleshort&0XFE0E     0x940C
>>>>>20     uleshort&0XFE0E     0x940C
>>>>>>24    uleshort&0XFE0E     0x940C
>>>>>>>28   uleshort&0XFE0E     0x940C
>>>>>>>>32  uleshort&0XFE0E     0x940C          AVR firmware

# RJMP (2 octets) pour Reset, Int0-2, PcInt0-3 and WDT
1           byte&0xF0           0xC0
>3          byte&0xF0           0xC0
>>5         byte&0xF0           0xC0
>>>7        byte&0xF0           0xC0
>>>>9       byte&0xF0           0xC0
>>>>>11     byte&0xF0           0xC0
>>>>>>13    byte&0xF0           0xC0
>>>>>>>15   byte&0xF0           0xC0
>>>>>>>>17  byte&0xF0           0xC0            AVR firmware
```

Évaluation de la signature. Cette signature a été testée avec des échantillons de micrologiciels provenant de ces projets libres : Marlin, QMK et les exemples de l'Arduino SDK. L'ensemble des échantillons est correctement détecté.

3.3 Contributions en upstream

Plutôt que de proposer un fork ou un nouvel outil, ces signatures ont été directement proposées par mail au mainteneur actuel de `file/libmagic`. À la date d'écriture de cet article, la dernière version (5.46) peut ainsi détecter les images binaires de micrologiciels ARM Cortex-M, AVR, Espressif ESP8266/ESP32, ainsi que certaines représentations alternatives telles que les formats Intel HEX (`.hex`) et Device Firmware Update (`.dfu`).

4 Identification des microcontrôleurs compatibles

Cette section présente une approche permettant d'identifier les modèles de microcontrôleurs ayant les périphériques nécessaires pour exécuter un micrologiciel donné. Cette méthode consiste à identifier les adresses lues et écrites dans l'espace mémoire destiné aux périphériques, puis de les corréler avec une base de données en source ouverte.

4.1 Identification des accès aux registres de périphériques

En désassemblant l'image du micrologiciel pour l'architecture identifiée (voir la section 3), il devient possible de trouver les instructions de lecture et d'écriture de la mémoire.

Dans le cas d'ARM Cortex-M. L'accès à un périphérique est généralement réalisé en deux temps : d'abord une adresse est chargée dans un registre par une première instruction de lecture mémoire `LDR`, puis ce registre est utilisé par une seconde instruction de lecture `LDR*` ou d'écriture `STR*` (voir Listing 8). En identifiant ces suites d'instructions avec un désassembleur, il est possible de collecter l'ensemble des adresses accédées en lecture ou en écriture. Parmi ces adresses, celles entre `0x40000000` et `0x5FFFFFFF` correspondent à des accès aux registres de périphériques.

Listing 8: Exemple d'écriture d'un GPIO en ARM Cortex-M

```
1 LDR   r3, [pc, #0x2c]  # pc+0x2c pointe le périphérique GPIOB
2 MOV.W r2, #0x800
3 STRH  r2, [r3, #0x18]  # 0x18 est l'offset du registre BSRR
```

En pratique, il existe quelques situations où il est difficile d'identifier ces suites d'instructions. Par exemple certains accès peuvent être réalisés en passant l'adresse de base d'un périphérique en argument d'une fonction. Pour garder l'implémentation simple, il est possible d'ignorer ces cas et d'accepter de ne pas être exhaustif.

4.2 Collecte des descriptions des registres de microcontrôleurs

Les fabricants distribuent des bibliothèques d'abstraction matérielle afin de faciliter l'adressage vers les différents périphériques de leurs microcontrôleurs. Certaines de ces bibliothèques sont partiellement générées à partir de descriptions, souvent au format XML. Notre but ici est donc de collecter les descriptions qui sont accessibles en source ouverte.[7]

Tableau 3. Formats de description de microcontrôleurs

Architecture	**Format de description**	**Exemples d'outils**
ARM Cortex-M	System View Description (SVD)	CMSIS, svdtools
AVR	AVR Tools Device File (ATDF)	avr-rust, atdf2svd
Risc-V	System View Description (SVD)[a]	riscv-rust
TI MSP430	DSLite (folder of XML files)	msp430_svd
Xtensa	System View Description (SVD)[a]	esp-rs

[a] Certains fabricants utilisent SVD bien qu'il s'agisse d'un standard ARM.

La majorité des descriptions sont distribuées au format System View Description (SVD), ou peuvent être facilement converties dans ce format (voir tableau 3). Le format SVD est un document XML avec un schéma défini par ARM. Un extrait de la description du NXP LPC54S005 (sous licence BSD-3-Clause) est présenté en exemple dans le Listing 9.

[7] Donc cette base n'aura pas de microcontrôleurs sous accord de non-divulgation.

Listing 9: Extrait de fichier System View Description (SVD)

```
<device schemaVersion="1.3">
  <name>LPC54S005</name>
  <peripherals>
    <peripheral>
      <name>SYSCON</name>
      <baseAddress>0x40000000</baseAddress>
      <registers>
        <register>
          <name>AHBMATPRIO</name>
          <addressOffset>0x10</addressOffset>
          <fields>
            <field>
              <name>PRI_ICODE</name>
              <description>I-Code bus priority.</description>
              <bitOffset>0</bitOffset>
              <bitWidth>2</bitWidth>
              <access>read-write</access>
            </field>
```

Sources agrégeant les descriptions. Chaque bibliothèque d'abstraction matérielle n'inclut en général que quelques descriptions SVD. Il n'est pas envisageable de manuellement collecter ces descriptions en installant manuellement de nombreux environnements de développement. Heureusement, il existe quelques sources les agrégeant.

Une première source communautaire est maintenue avec le module Python `cmsis-svd`, disponible sur GitHub à `https://github.com/cmsis-svd/cmsis-svd-data`. Cette base contient plus de 2 000 descriptions SVD provenant d'une vingtaine d'intégrateurs.

Une seconde source est maintenue par ARM, accessible sur `https://www.keil.arm.com/devices/`. Cette collection est directement alimentée par une trentaine d'intégrateurs et permet à l'environnement de développement ARM Keil de supporter leurs microcontrôleurs.

Mise en forme de la base de données. Les fichiers XML collectés ne sont pas structurés dans un format optimal permettant de filtrer à partir des adresses de registres. Nous implémentons un script Python afin de convertir un dossier de descriptions SVD vers une base de données SQLite (voir la figure 4). Ce script permet de passer de 15 Gio de SVD provenant des sources précédentes à une base de données de 315 Mio.

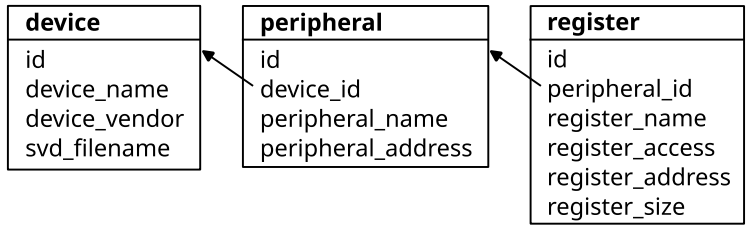

Fig. 4. Schéma de la base de données SQL

4.3 Recherche des microcontrôleurs correspondants

Un script Python `chiprec.py` est proposé. Il se restreint au cas d'ARM Cortex-M et implémente l'identification des adresses de registres puis la recherche des microcontrôleurs compatibles dans la base de données.

La recherche des microcontrôleurs est effectuée par élimination. L'algorithme commence en considérant tous les microcontrôleurs comme étant compatibles. Ensuite, pour chaque adresse accédée par le micrologiciel, les microcontrôleurs n'ayant pas un périphérique avec un registre à cette adresse sont éliminés.

Une fois la recherche terminée, l'outil présente la liste des microcontrôleurs compatibles en explicitant le nom des périphériques utilisés.

Démonstration sur un micrologiciel libre. Nous lançons l'outil sur le micrologiciel Betaflight, compilé pour le microcontrôleur STM32H750 (voir Listing 10). L'outil identifie plusieurs microcontrôleurs de la même famille `STM32H7`, cela s'explique par la similarité entre ces microcontrôleurs.

Listing 10: Extrait de l'exécution de `chiprec.py` sur un micrologiciel

```
$ ./chiprec.py ./betaflight_4.4.2_STM32H750.bin
Found addresses: 0x58024428 (read), 0x58024454 (read) [...]

STM32H742x (STM32H742x.svd):
[...]

STM32H753 (STM32H753.svd):
    read register PLLCKSELR of RCC
    read register D2CCIP2R of RCC
    read register D3CCIPR of RCC
    read register DIER of TIM6
    read register CR of CRS
    read register D3CR of PWR
    read register CR of RCC
    [...]

STM32H755_CM4 (STM32H755_CM4.svd):
[...]
```

Références

1. Ian Darwin and Christos Zoulas. file command and the libmagic library. `https://www.darwinsys.com/file/`.
2. Yanick Fratantonio, Elie Bursztein, Luca Invernizzi, Marina Zhang, Giancarlo Metitieri, Thomas Kurt, Francois Galilee, Alexandre Petit-Bianco, and Ange Albertini. Magika content-type scanner. `https://github.com/google/magika`.
3. Louis Granboulan. cpu_rec.py, un outil statistique pour la reconnaissance d'architectures binaires exotiques. *SSTIC*, 2017.
4. Craig Heffner and ReFirmLabs. binwalk : Firmware analysis tool. `https://github.com/ReFirmLabs/binwalk`.
5. Guillaume Heilles and Damien Cauquil. binbloom : raw binary firmware analysis software. `https://github.com/quarkslab/binbloom`.
6. Arm Limited. Cortex-m3 devices generic user guide, memory model. `https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/memory-model`.
7. Marco Pontello. Trid - file identifier. `https://mark0.net/soft-trid-e.html`.
8. Ville Skyttä and David Faure. Shared mime info. `https://www.freedesktop.org/wiki/Software/shared-mime-info/`.
9. Evan Sultanik and Trail of Bits. Polyfile : Python cleanroom implementation of libmagic. `https://github.com/trailofbits/polyfile`.