Rétro-ingénierie de systèmes embarqués AUTOSAR

Etienne Charron et Axel Tillequin
etienne.charron@renault.com
axel.tillequin-extern@renault.com

Renault

Résumé. On présente une introduction à l'étude d'équipements automobiles, plus particulièrement des systèmes embarqués respectant le standard AUTOSAR [1] Classic. Après une brève présentation du contexte et des spécificités de ces systèmes, on présente plusieurs approches permettant de faciliter l'analyse statique de ces systèmes. Ces approches visent à identifier certaines fonctionnalités pour permettre de se focaliser sur la recherche de vulnérabilités des services exposés, en automatisant la recherche des fonctions de diagnostics (UDS), différents "modules" AUTOSAR et/ou des structures utilisées par l'OS permettant d'identifier les applications. Cette dernière approche peut s'étendre à l'étude d'autres systèmes embarqués comme par exemple ceux basés sur FreeRTOS, Zephyr, etc.

1 Introduction

Dans le domaine automobile, le développement des services connectés permet par exemple d'interagir avec une voiture depuis son smartphone, cette surface d'exposition grandissante nécessite donc quelques précautions.

Bien que l'architecture d'un véhicule (voir Fig.1) tend à cloisonner autant que possible les composants exposés communiquant via un réseau ethernet (ainsi les interfaces "ouvertes" comme la prise de diagnostic ou la prise de charge électrique), des composants plus "critiques", les scénarii d'exploitation continuent d'exister [8]. Il est nécessaire de s'intéresser à la défense en profondeurs, en analysant non seulement les équipements exposés (boîtier multimédia, télématique), mais aussi les équipements moins exposés mais sensibles.

Les équipements communiquant sur le bus CAN sont généralement des systèmes temps réels spécifiques au monde automobile respectant le standard AUTOSAR [2]. Ce standard populaire permet notamment d'assurer une certaine interopérabilité entre les équipements.

[1] AUTomotive Open System ARchitecture

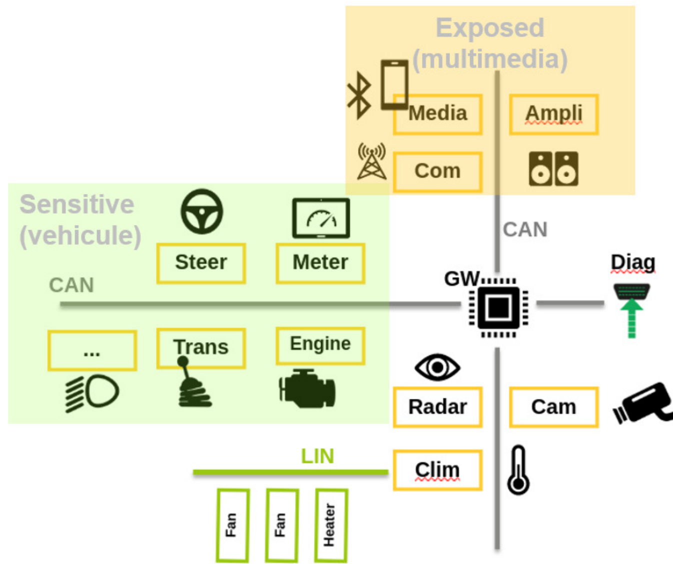

Fig. 1. Architecture d'un véhicule

L'analyse de ces systèmes, en particulier leur rétro-ingénierie par analyse statique, pose plus de difficultés que celle d'un système de type Unix/Linux en raison de l'absence de base de connaissance sur le fonctionnement interne des différents OS propriétaires implémentant ce standard.

Notre objectif est de montrer ici comment initier rapidement l'analyse statique d'un tel système. On s'appuie sur une analyse réalisée sur un système central du véhicule, implémenté sur une architecture AURIX exploitant quatre cœurs Tricore.

2 Contexte

2.1 Prérequis

Pour analyser un tel système, on utilise soit une image du *firmware* issue d'un paquet de mise à jour, une image obtenue par *dump* de la mémoire dans le cas ou le JTAG est accessible ou enfin une image extraite d'une mémoire flash externe.

Évidemment, à la différence d'un firmware au format ELF, ce firmware est généralement un simple "blob" dépourvu de dépendances externes et d'indications de segments de code ou de données. Comme pour la plupart des analyses de firmware de systèmes embarqués, on doit donc avant tout

déterminer la bonne localisation du firmware en mémoire. Un peu d'huile de coude ou un outil comme *binbloom* [4] peuvent être utiles.

A ce stade, une première auto-analyse par des outils comme *IDA Pro* ou *Ghidra* permet de distinguer des zones de code et des zones de données. Pour confirmer ou infirmer la bonne localisation du firmware on peut se reporter à la datasheet du microcontrôleur et voir si l'usage de registres/adresses associés au matériel (timers, etc) semble cohérent.

Pour simplifier, on fait abstraction des problématiques de "double-banking" (duplication du firmware) et de relocalisation dynamique de certaines zones de code (similaire à une forme d'unpacking) et on considère donc qu'on dispose d'une image bien mappée comparable à un *dump* JTAG complet du firmware.

2.2 Observations

L'analyse d'une telle image de *firmware* présente plusieurs particularités par rapport à celle d'un système plus classique de type Unix/Linux. D'abord, ce système temps-réel est le plus souvent dénué de notion de fichier et dépourvu de chaînes de caractères... Deux éléments qui fournissent habituellement des indices utiles sur l'organisation et la nature du code exécuté.

Dans *Ghidra*, on se retrouve typiquement face à plusieurs milliers de fonctions "anonymes". Ces fonctions implémentent l'ensemble des couches du standard AUTOSAR, mais en dehors de celles très bas-niveau manipulant directement des registres/adresses associés au matériel il est difficile de cibler directement un jeu de fonctions associé à une fonctionnalité de haut niveau comme par exemple la manipulation d'un payload applicatif reçu d'un autre équipement. On se retrouve donc très vite noyé, sans indicateurs simples permettant de séparer les fonctions internes de l'OS de celles des couches d'applications.

On sait néanmoins que le système respecte différents standards automobiles comme l'UDS [7] et le standard AUTOSAR. On va donc maintenant chercher à s'appuyer sur ces standards pour identifier certaines parties du système.

3 Utilisation des standards automobiles pour retrouver les symboles

3.1 Architecture AUTOSAR

Le standard AUTOSAR, dans sa forme dite "Classic" qui adresse les besoins de temps réels forts et de sûreté de fonctionnement, se décompose

en plusieurs modules (voir Fig.2) qui décrivent l'architecture du système d'exploitation et son middleware. L'ensemble forme se qu'on appelle le BSW (Basic SoftWare). L'interface entre ce BSW et la couche applicative est le RTE (RunTime Environment).

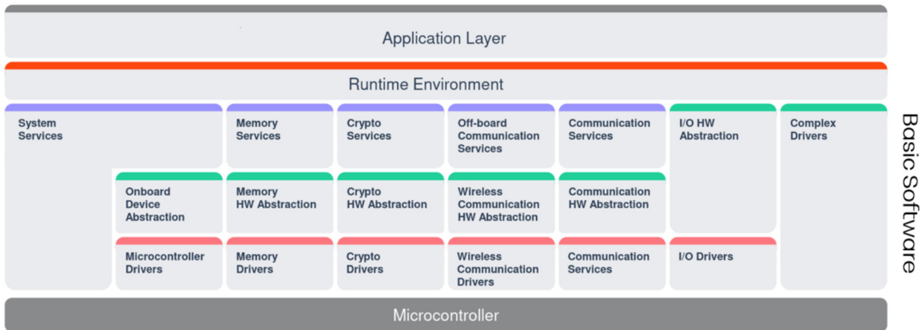

Fig. 2. Architecture d'un système AUTOSAR

L'OS fait partie des "System Services" et sa spécification est issue du standard OSEK/VDX, comparable au standard POSIX du monde Unix/Linux. On y décrit la notion de tâche, d'ordonnanceur de ces tâches, la gestion des interruptions, etc. En pratique les implémentations de ce type d'OS compatible AUTOSAR se comptent sur les doigts d'une main... et sont des solutions propriétaires (Vector, Elektrobit, etc [3]) partagées uniquement avec des fournisseurs (Bosch, Continental, Valéo, etc) chargés des développements des autres modules du BSW, voire aussi la couche applicative.

Au-delà de la spécification des modules, le standard décrit aussi la méthodologie à suivre pour générer un firmware qui contiendra uniquement les modules nécessaires et dont le RTE aura été généré pour fournir une interface minimale avec le BSW. En conséquence, l'implémentation effective du BSW et du RTE peut être sensiblement différente d'un composant à un autre. Enfin, une difficulté supplémentaire est que ce standard est en constante évolution est que l'API de certains modules varie sensiblement avec les versions.

3.2 Identification des fonctions via les codes d'erreur DET

Le module DET (Default Error Tracer) [1] décrit une interface permettant au développeur de remonter des codes d'erreurs à des fins de debug.

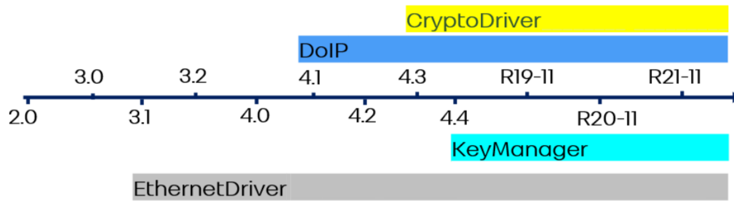

Fig. 3. Évolution du standard AUTOSAR

Ce module n'est supposé être présent que dans les firmwares en cours de développement, mais il est fréquemment conservé dans les versions de production et ces fonctions sont facilement identifiables dans le firmware, car elles ont typiquement beaucoup de références croisées et un prototype très reconnaissable.

Listing 1: Prototype des fonctions DET

```
1 Det_ReportError( uint16 ModuleId, uint8 InstanceId,
2                  uint8 ApiId, uint8 ErrorId )
3 Det_ReportRuntimeError( uint16 ModuleId, uint8 InstanceId,
4                         uint8 ApiId, uint8 ErrorId)
```

Chaque module AUTOSAR possède un numéro d'identifiant (voir Fig.4), ainsi que ses principales routines (ou "services"). On peut donc se servir de ces différents identifiants, en particulier celui qui caractérise chaque module pour identifier le module probable auquel appartient une fonction qui remonte une erreur. Dans certains cas, on peut même identifier précisément le symbole et le prototype exact de ces fonctions.

Par exemple, le service "`TcpIp_TcpListen`" est identifié par le `module id` 170 (TCP/IP Stack) et le `service id` 0x07 (voir Fig.4).

Pour chaque fonction faisant appel à `Det_ReportError`, on peut donc automatiser [2] l'analyse du code décompilé, reconnaître les identifiants utilisés et ainsi renommés la fonction avec le nom du module et/ou du service. Au préalable on aura évidemment extrait les numéros d'identifiants de l'ensemble des PDF du standard AUTOSAR.

Évidemment, dans la vraie vie tout n'est pas si simple. En effet, la fonction parente ayant levée l'erreur n'est pas nécessairement la fonction directement concernée par cette erreur. Il peut s'agir d'une erreur qui concerne :

[2] ps : les scripts *Ghidra* seront publiés

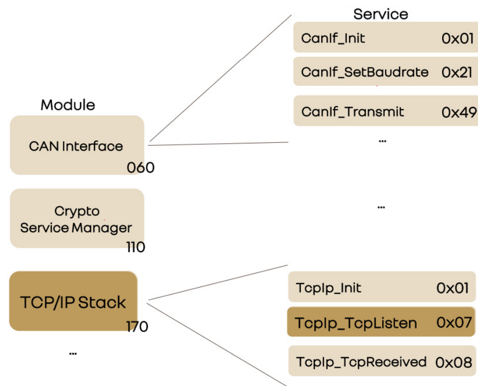

Fig. 4. Identifiants des modules et services

— un autre service dont dépend la fonction (voir Listing 2), auquel cas il faut inspecter le code d'erreur ou utiliser une autre approche,
— une autre fonction placée en amont dans la pile d'appels (voir Listing 3), auquel cas il est probable que notre fonction fasse néanmoins partie du module et on peut au minima préfixer son nom.

Il est nécessaire de discriminer certains cas, notamment en vérifiant le nombre de paramètres de la fonction, et quand c'est possible leurs types (la convention d'appel du Tricore permet de différencier les paramètres de type entier et de type pointeur suivant le registre utilisé).

Listing 2: DET : Renomage d'une function parente levant l'erreur

```
1  void TcpIp_GetVersionInfo(Std_VersionInfoType *info) {
2    if (DAT_5000e6ca == '\0') {
3      Det_ReportError(1,0xaa,1,2);
4      return;
5    }
6    if (info == (ech_Std_VersionInfoType *)0x0) {
7      Det_ReportError(1,0xaa,2,2);
8      return;
9    }
10   info->vendorID = 0x1e;
11   info->moduleID = 0xaa;
12   info->instanceID = '\x0f';
13   info->sw_major_version = '\0';
14   info->sw_minor_version = '\0';
15   return;
16 }
```

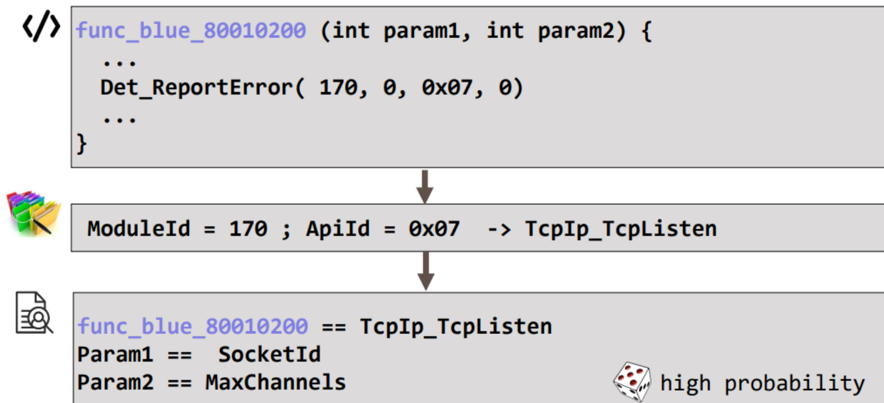

Fig. 5. Identification du service `TcpIp_TcpListen` grâce aux identifiants DET

Listing 3: DET : Renomage d'une function présente dans la fonction parente levant l'erreur

```
1  int FUN_8013a740(void) {
2    int iVar1;
3
4    iVar1 = SchM_Enter_New(3);
5    if ((iVar1 == 1)) {
6      Det_ReportError(2,0,3,5);
7    }
8    [...]
9    iVar1 = SchM_Exit_New(3);
10   if (iVar1 == 1) {
11     Det_ReportError(2,0,4,5);
12   }
13   [...]
14 }
```

3.3 Identification de fonction de diagnostics (UDS)

Présentation de l'UDS Il est intéressant d'identifier rapidement les fonctions manipulant des données externes. Généralement les équipements automobiles exposent à minima des services diagnostics pour des besoins de maintenance (mise à jour, de calibration, etc). La plupart du temps ces services sont implémentés en suivant le standard UDS défini par l'ISO 14229-1 et exposé sur le bus CAN ou de plus en plus également sur ethernet

via le protocole DoIP (Diagnostics Over IP). L'UDS propose une multitude de services que l'équipement peut implémenter ou de ne pas implémenter suivant les besoins.

Identification des services implémentés les services peuvent être listés dynamiquement avec diffèrent outils comme *CANanalyze* [5], ou *Scapy* [10]. Par exemple, (voir Listing 4) on peut obtenir les services UDS supportés par un équipement de la façon suivante :

Listing 4: Détection dynamique des services UDS (*CANanalyze*)

```
python nmap.py A komodo 0x18daf1d4 0x18dad4f1 services
scan.services discovered 10 Diagnostic Session Control
scan.services discovered 14 Clear Diagnostic Information
scan.services discovered 19 Read DTC Information
scan.services discovered 22 Read Data By Identifier
scan.services discovered 27 Security Access
scan.services discovered 2e Write Data By Identifier
scan.services discovered 31 Routine Control
scan.services discovered 3e Tester Present
```

Tous les services ne représentent pas le même intérêt d'un point de vue cybersécurité, sur l'exemple (Listing 4) seulement les services `Write Data by Identifier` et `Security Access` semblent intéressants. Le premier manipule des données applicatives, le second permet de s'authentifier pour effectuer des actions privilégiées.

Identification du service `Read Data By Identifier` Ce service permet de récupérer des données sauvegardées sur l'équipement (version du logiciel, version matérielle, VIN [3]).

Chaque donnée est associée à un identifiant (DBI) dont certains sont fixés par le standard UDS (cf. tableau 1) :

Ce service ne nécessite pas d'être authentifié, et bien que ne présentant pas une surface d'attaque très importante, le fait d'identifier son implémentation dans le firmware est utile simplement pour trouver aussi d'autres services UDS comme le `Security Access` ou éventuellement les `Routine Control`.

[3] VIN : Vehicle Identification Number

DBI	Contenue
...	...
F180	Supplier Identifier
F190	VIN
F193	Hardware Version Number
F195	Software Version Number
...	...

Tableau 1. Exemple d'identifiants standardisés par l'UDS [7]

Listing 5: Détection dynamique des services UDS

```
1  python nmap.py A komodo 0x18daf1d4 0x18dad4f1 dbis
2  ...
3  scan.dbis read  DBI F195 : [0x44, 0x39, 0x30, 0x33, 0x31, 0x31] ->
   ↪  "D90311"
4  ...
5  scan.dbis discovered :[0x100, 0x111, 0x112, 0x200, 0x210, 0x211,
6      ...
7      0xf058, 0xf05a, 0xf05c, 0xf05d, 0xf060, 0xf0a6, 0xf0d2,
8      0xf182, 0xf187, 0xf188, 0xf18a, 0xf18c, 0xf18e, 0xf190,
9      0xf191, 0xf194, 0xf195, 0xf1a0, 0xf1a1, 0xf1a2, 0xf1f3,
10     0xf1f4, 0xf1f5, 0xf1f6, 0xf1f7, 0xfd01, 0xfd10, 0xfd12,
11     0xfd14, 0xfd15, 0xfd20]
12 scan.dbis writeable discovered :[]
```

En utilisant ce service (voir Listing 5) il est possible de rechercher des valeurs particulières dans le firmware afin de retrouver les fonctions du service `Read Data By Identifier` par analyse de références croisées sur les réponses obtenues.

Généralement les DBI sont accessibles en lecture/écriture par un "getter" (respectivement "setter"), cette approche permet d'identifier et de renommer un grand nombre de fonctions, mais surtout d'identifier ce service UDS facilement. Ici, le DBI `F195` est associé à une des rares chaînes de caractère présentes dans le firmware "D90311". La recherche conjointe de ces valeurs permet de localiser dans le firmware la table des structures de *handlers* de chaque DBI contenant typiquement les pointeurs vers les fonctions "getter" et "setter" associées.

Identification des autres services UDS D'un point de vue développement, lorsqu'un équipement reçoit une requête de diagnostic, celle-ci est analysée par une fonction, puis distribuée à la fonction implémentant le service en question. Il suffit donc généralement d'identifier un premier

service pour identifier tous les services en analysant les références croisées comme le montre la Fig.6.

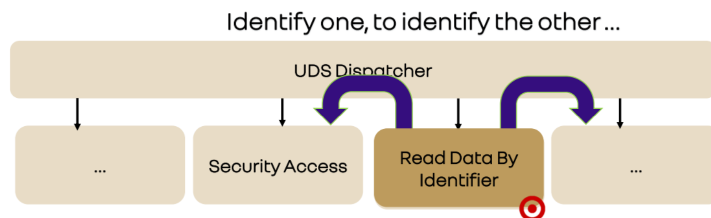

Fig. 6. UDS Dispatcher : Identification d'un service, pour les identifier tous

On cherche donc en particulier la fonction `UDS_Dispatcher` qui se présente généralement comme un grand *switch-case*.

4 Analyse statique par identification de structures de données

Les approches précédentes s'appuient sur l'utilisation d'informations provenant de standards pour identifier certaines fonctions du firmware. L'approche proposée ici se base sur l'utilisation de l'outil *Ccrawl* [9] pour effectuer la collecte de types structurés, en particulier de définitions struct/union/class dans des fichiers sources C/C++, puis la reconnaissance de "patterns" caractéristiques dans le firmware et ainsi identifier la présence ou l'usage de ces structures.

4.1 Mapping des registres matériels

Dans le standard AUTOSAR, on a vu que les modules de type *drivers* ont une adhérence forte à l'architecture matérielle et on peut donc évidemment envisager d'en identifier les fonctions par références sur les registres/adresses dédiés tels que décris par la datasheet du microcontrôleur.

Le plus souvent cette datasheet est disponible publiquement, et il n'est pas rare que le fondeur fournisse également un ensemble de *headers* spécifique pour chaque version du SoC [4] permettant aux développeurs de manipuler plus facilement ces registres afin de configurer le microcontrôleur

[4] System on a Chip

et les différentes interfaces CAN, ETH, le GPT (General Purpose Timer), le contrôleur d'accès direct à la RAM (DMA), ou d'interagir avec un HSM.

Dans le cas d'un SoC AURIX utilisant un ou plusieurs microcontrôleurs Tricore, on peut obtenir plusieurs exemples [6] de code bas-niveau et l'ensemble des headers C décrivant les structures à mapper aux bonnes adresses.

Listing 6: Exemple d'interface hardware avec le "Flexible CRC Engine" d'une carte AURIX

```
\$ ccrawl -l aurix.db collect Libraries/iLLD/TC38A/Tricore/
    [...]
\$ ccrawl -l aurix.db show -f C "struct _Ifx_FCE"
struct _Ifx_FCE {
  Ifx_FCE_CLC CLC;
  Ifx_UReg_8Bit reserved_4[4];
  Ifx_FCE_ID ID;
  Ifx_UReg_8Bit reserved_C[20];
  Ifx_FCE_CHSTS CHSTS;
  Ifx_UReg_8Bit reserved_24[200];
  Ifx_FCE_KRSTCLR KRSTCLR;
  Ifx_FCE_KRST1 KRST1;
  Ifx_FCE_KRST0 KRST0;
  Ifx_FCE_ACCEN1 ACCEN1;
  Ifx_FCE_ACCEN0 ACCEN0;
  Ifx_FCE_IN IN[8];
};
```

Sauf dans les cas rares où *Ghidra* (ou *IDA Pro*) supporte déjà la définition précise du SoC, il est nécessaire de pouvoir mapper la définition de ces registres "hardware" en mémoire si l'on s'intéresse à l'identification des différents drivers.

Les outils comme *Ghidra* sont normalement capables d'importer des définitions à partir d'un jeu de fichiers sources en C ou C++. Toutefois, cette opération nécessite le plus souvent de connaître précisément l'ensemble des macro-définitions nécessaires lors de la compilation. En pratique, lorsqu'il s'agit de sources correspondant à du code bas-niveau et visant une architecture tierce, réussir l'import relève plutôt du miracle.

Une alternative est proposée par l'outil *Ccrawl* qui permet de propager une définition vers *Ghidra*. Dans l'exemple de la Fig.7, une fois la structure `struct _Ifx_FCE` importée et mappée à la bonne adresse, le code décompilé des fonctions associées devient un peu plus lisible.

```
In [1]: from ccrawl.ext.ghidra import *
In [2]: s = db.rdb.db["nodes"].find_one({"id": "struct Ifx_FCE"})
In [3]: x = ccore.from_db(s)
In [4]: build(x,db)
building data type struct _Ifx_FCE...
```

```
void FUN_80326e00(void) {
[...]
uVar1 = FUN_801c66e2(_DAT_f003600);
 _DAT_f0000000 = 0;
[...]
_DAT_f0000118 = 0xffffffff;
 _DAT_f0000108 = _DAT_f0000108 &
                0xfff0ffff;
 [...]
}
```

```
void FUN_80326e00(void) {
[...]
pw = bsw_get_con0_pw(SCU.WDTCPU+1);
CRC_Engine.CLC = 0;
[...]
CRC_Engine.IN[0].CRC = 0xffffffff;
CRC_Engine.IN[0].CFG = CRC_Engine.IN[0].CFG &
                      0xfff0ffff;
 [...]
}
```

Fig. 7. Ccrawl : Exportation d'une structure vers Ghidra

4.2 Localisation de structures globales de l'OS

Le fait de constituer une base de données des définitions de structures (et plus généralement de types) permet aussi de capitaliser un savoir-faire au cours de la rétro-ingénierie des différents systèmes que l'on rencontre. En particulier, l'accès même incomplet à des bouts de code source permet d'accumuler des informations sur les structures internes de l'OS.[5]

Pour des systèmes temps-réels comme le cas d'un OS AUTOSAR, ces structures internes sont le plus souvent statiques et fixées au moment de la compilation du firmware. C'est le cas pour plusieurs entités de l'OS comme les applications, les taches, les alarmes, et toutes les sous-structures associées (descripteurs de pile, etc). Ces structures présentent souvent des "cycles" de références : par exemple, une tâche va être associée à la structure d'un cœur d'exécution qui lui-même possède une référence vers la liste des tâches qu'il exécute.

Afin de les localiser dans le firmware, on peut calculer la "signature" d'un cycle de référence. L'outil *Ccrawl* permet de calculer ces signatures pour une structure racine donnée. On peut par exemple obtenir un graphe de dépendance entre structures de la forme suivante (ici sur un exemple anonymisé) : la structure `struct grG` possède un champ `pB` de type pointeur de pointeur vers une structure `struct grB` qui elle-même possède un champ de type pointeur vers `struct grG` qui se trouve être la structure d'origine.

En pratique, une signature est une simple liste comme dans l'exemple du Listing 7 qui provient d'un OS AUTOSAR très courant et qui indique qu'à l'offset 32 on trouve un champ `IdleTask` qui se trouve être un

[5] de ce point de vue, *github* est une mine d'or à ciel ouvert...

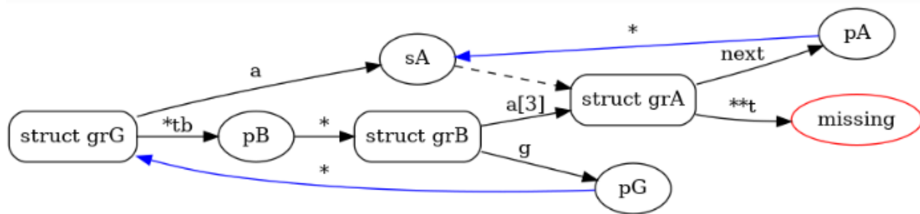

Fig. 8. Ccrawl : Exemple de graphe de dépendance entre structures avec cycles

pointeur, on suit ce pointeur qui à son offset 0 contient une sous-structure `Thread` qui à son offset 20 contient un champ `Core` qui est un pointeur vers la structure d'origine.

Listing 7: Signature pour `Os_CoreAsrConfigType_Tag`

```
sig_OS_CoreAsrConfigType_Tag = [
  [(32, 'IdleTask'),
   '*',
   (0, 'Thread'),
   (20, 'Core'),
   '*'
  ]
]
```

À partir de cette définition, il est possible de parcourir automatiquement le firmware à la recherche des structures statiques correspondantes. Par exemple en utilisant le script *Ghidra* fourni en Annexe dans le Listing 8.

4.3 Identification de structures dans les fonctions

De façon plus générale, il est utile de pouvoir identifier l'usage de certaines structures dans les fonctions du firmware. Par exemple on souhaite pouvoir automatiser l'identification d'une structure AUTOSAR simple comme dans le cas suivant :

Sur la Fig.9 l'approche par code d'erreurs DET permet déjà d'identifier cette fonction, mais vu la "forme" de la structure `std_VersionInfoType` il semble possible également de l'identifier par le fait qu'elle (dé)référence les champs de cette structure.

Ainsi, pour une fonction donnée, en s'appuyant sur les fonctionnalités de *Ghidra*, *Ccrawl* permet (voir Fig.10) :

Fig. 9. Ccrawl : Identification d'une structure dans une fonction

— de déterminer les pointeurs, (passés en arguments ou variables locales) qui sont déréférencés à des offsets particuliers,
— puis de chercher les définitions de structures ayant ces offsets dans la base de donnée utilisée.

Inversement, pour une structure donnée, on peut rechercher les fonctions qui semblent utiliser des pointeurs sur ce type de structure (voir Fig.11) :

5 Conclusion

Nous avons présenté plusieurs approches permettant d'initier rapidement l'analyse d'un firmware de système embarqué automobile basé sur le standard AUTOSAR.

```
In [1]: from ccrawl.ext.ghidra import *

In [2]: find_auto_structs('FUN_80082932')
Out[2]: {'param1': [(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)]}

In [3]: L = _

In [4]: db.rdb.find_matching_types(L, psize=4)

In [5]: L
Out[5]: {'param1': ([(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)],
          ['struct Std_VersionInfoType',
           'struct ?_22ecf778',
           'struct EVTtstrInternalEntries'])}
```

Fig. 10. Ccrawl : recherche d'une définition de structure dans une fonction

```
In [1]: from ccrawl.ext.ghidra import *

In [2]: s = db.rdb.db["nodes"].find_one({"id": "struct Std_VersionInfoType"})
In [3]: x = ccore.from_db(s)
In [4]: ax = build(x,db)
In [5]: ax().offsets(psize=4)
Out[5]: [(0, 2), (2, 2), (4, 1), (5, 1), (6, 1)]}
In [6]: find_functions_with_type(_,sta=0x80080000,sto=0x80090000)
Out[6]: [FUN_80082932, FUN_8008341e, …]
```

Fig. 11. Ccrawl : recherche des fonctions ayant un pointeur sur une structure donnée

Les deux premières, par identification des fonctions via le module DET [6] et via les services UDS,[7] sont facilement automatisables. La première s'appuie sur une base d'identifiants extraits des spécifications du standard AUTOSAR et peut s'appliquer simplement après l'auto-analyse du firmware par un outil comme *Ghidra*. Le seconde s'appuie sur la localisation dans le firmware des identifiants (DBI) du service `Read Data by Identifier` de l'UDS, voire des valeurs renvoyées par ce service dans le cas ou l'on peut faire une détection dynamique de ces services et identifiants.

Nous avons appliqué ces deux approches sur quelques firmwares issus du monde automobile. Le tableau ci-dessous décrit le pourcentage de fonctions retrouvé. Il est important de noter que le cas du firmware n°2 n'est pas problématique : la méthode DET n'est ici pas applicable, mais ce firmware transmet beaucoup d'information via une interface UART il

[6] Default Error Tracer
[7] Unified Diagnostic Services

est possible de réduire les fonctions en analysant directement les chaînes de caractères qu'il contient.

.

	Chaînes de caractères	% DET [8]	% DBI [9]
Firmware 1	15	10%	8%
Firmware 2	5472	0%	1%
Firmware 3	361	2%	4%

Tableau 2. Légende du tableau

La troisième approche, par identification de structures dans une base de données que l'on construit au fur et à mesure des analyses et des accès dont on dispose à certains codes sources, est une aide précieuse pour améliorer la décompilation, mais nécessite plus d'interaction avec l'analyste. L'importation des structures décrivant les registres hardware permet d'identifier les fonctions des drivers, l'outil Ccrawl permet de localiser des structures globales décrivant les tâches ou les alarmes de l'OS (si cet OS est connu dans la base de données). Toutefois, la précision des résultats dépend des caractéristiques de la structure recherchée. Il est clair que plus une structure est de taille importante et possède des champs de tailles variées plus sa détection sera précise. En pratique, sur notre cas d'étude on retrouve l'ensemble des structures internes de l'OS décrivant les cœurs d'exécution, et par conséquent l'ensemble des structures décrivant les applications et tâches qu'il doit exécuter.

Annexes

Listing 8: Recherche de signatures de structures dans Ghidra

```
def find_struct_by_sig(sigs,start,stop,step=4):
    R = []
    for cur in range(start,stop,step):
        prob = 0.0
        for c in sigs:
            if check_cycle(c,address=cur):
                prob += 1./len(sigs)
        if prob>0.2:
            R.append((cur,prob))
    return R

def check_cycle(c,address):
    if len(c)==0:
        return False
    x = address
    for y in c:
        if y=='*':
            try:
                x = getInt(toAddr(x))
                if x<0:
                    x = 1+(0xffffffff+x)
            except:
                x = None
                break
        else:
            x += y[0]
    return x==address
```

Références

1. AUTOSAR. Specification of Default Error Tracer, AUTOSAR CP R22-11. https://www.autosar.org/fileadmin/standards/classic/22-11/AUTOSAR_SWS_DefaultErrorTracer.pdf, 2022.
2. AUTOSAR. Standard. https://www.autosar.org/standards/classic-platform, 2022.
3. AUTOSAR. Vendor ID. http://web.archive.org/web/20220613202339/https://www.autosar.org/about/vendorid/, 2022.
4. Damien Cauquil. Binbloom v2. https://www.sstic.org/2022/presentation/binbloom_v2.
5. Etienne Charron Erwan Le Disez. CANanalyze : a python framework for automotive protocols. https://www.sstic.org/2020/presentation/cananalyze__a_python_framework_for_automotive_protocols/, 2020.

6. Infineon. AURIX code examples. https://github.com/Infineon/AURIX_code_examples, 2022.
7. ISO. 14229-1, 2020.
8. Synacktiv. 0-click RCE on the Tesla Model3. https://www.synacktiv.com/sites/default/files/2022-10/tesla_hexacon.pdf, 2022.
9. Axel Tillequin. Ccrawl. https://github.com/bdcht/ccrawl, 2022.
10. Nils Weiss. Scapy Automotive-specific documentation. https://scapy.readthedocs.io/en/latest/layers/automotive.html, 2022.