cpu_rec.py

Louis Granboulan

Airbus – prénom.nom@airbus.com

1 Pourquoi : reconnaissance automatique de CPU

Lors d'une étude de la sécurité d'un produit incluant du logiciel, il est utile de pouvoir faire une rétro-ingénierie des logiciels inclus, en particulier des codes binaires (fichiers exécutables ou bibliothèques). Pour pouvoir désassembler un code binaire, il est nécessaire de savoir à quel microprocesseur il est dédié (savoir quel est l'ISA *Instruction Set Architecture*).

Ce besoin est souvent rencontré lors de nos audits, et dans le cadre du Pré-GDR Sécurité Informatique, Airbus Group Innovations a proposé ce sujet pour REDOCS2016 [1], en fournissant comme élément de départ un corpus de codes binaires pour différentes architectures. Dans ce cadre de REDOCS2016, plusieurs pistes ont été explorées ; j'en utilise ici une autre, qui aboutit à un outil fonctionnel.

La plupart des codes binaires sont inclus dans des conteneurs (PE, ELF, Mach-O, COFF) qui précisent quelle est l'architecture cible et où sont les instructions pour le microprocesseur, mais de nombreux code embarqués utilisent des formats de fichiers non standards, et ces informations ne sont pas facilement disponibles.

Pour identifier quelle est l'architecture, une approche est d'utiliser un désassembleur (par exemple IDA) en testant toutes les architectures connues de cet outil et en devinant si le résultat du désassemblage a une signification. L'outil `cpu_rec.py` permet d'éviter cette approche fastidieuse en indiquant quelle est l'architecture et où se trouve la la section de code exécutable. Comme l'outil `cpu_rec.py` utilise des méthodes statistiques, ses résultats ne sont pas garantis et il est nécessaire d'utiliser ensuite un désassembleur pour affiner le résultat ; mais il n'est plus nécessaire de lui faire tester toutes les architectures.

1. Rencontres Entreprises DOCtorants Sécurité `http://confiance-numerique.clermont-universite.fr/redocs2016/`, dans le cadre duquel ont travaillé sur ce sujet Sebanjila Kevin Bukasa, Benjamin Farinier, Omar Jaafor et Nisrine Jafri.

2 Comment : analyse statistique avec fenêtres glissantes

2.1 Analyse statistique

Il existe de nombreuses familles de microprocesseurs, très différentes les unes des autres. Chaque microprocesseur inclut un module de décodage, qui prend en entrée les instructions pour les découper et les envoyer dans les unités de calcul appropriées. Que ce soit un microprocesseur à instructions de taille fixe (typiquement 4 octets pour un grand nombre de processeurs RISC) ou à instructions à tailles variables (typique des processeurs CISC, dont les plus emblématiques sont la famille Intel x86), le décodage dépend de la valeur du premier octet, dont on déduit la signification des octets suivants (ou l'absence d'octet suivant, si l'instruction est de longueur 1).

Cette façon de fonctionner des modules de décodage donne donc à chaque architecture (ISA) une signature statistique : étant donnée la valeur du premier octet d'une instruction, la distribution des octets suivants n'est pas uniforme. La signature statistique du code binaire dépend aussi de la structure du programme, engendré par un compilateur ou manuellement, car certaines séquences d'instructions ont des probabilités plus élevées (e.g. prologues ou fins de fonctions) et certaines instructions incluent des constantes numériques dont la distribution de probabilité dépend du programme.

L'apprentissage d'une signature statistique est au cœur des approches de type *machine learning*. Un moyen de tester de nombreuses techniques d'apprentissage statistique est d'utiliser la bibliothèque `scikit-learn`.

En pratique, un code binaire sera interprété comme une suite d'octets, qui s'analyse naturellement avec des approches dérivées de *bag of words*[2]. L'apprentissage statistique marche d'autant mieux que le corpus sur lequel est fait l'apprentissage est de grande taille. En expérimentant avec `scikit-learn`, la conclusion est que pour les CPU exotiques le corpus de départ (celui de REDOCS2016) est trop petit pour permettre une méthode plus efficace que *Multinomial Naive Bayes* sur des n-grammes[3]. Cf. plus de détails en annexe A.

2. Le `https://en.wikipedia.org/wiki/Bag-of-words_model` a été développé pour l'analyse de textes et la classification de documents, et est donc l'approche naturelle si on considère qu'un ISA est un langage

3. Le tutoriel `http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html` est une bonne introduction aux outils de classification de texte présents dans scikit-learn

Cette limitation du corpus de départ impose de trouver un moyen de mesurer la confiance en le résultat du classifieur, ce qui permettra d'une part de faciliter la détection d'architectures inconnues, et d'autre part d'éliminer du corpus des données invalides qui perturbent la classification. La technique *Multinomial Naive Bayes* est connue pour être un bon classifieur mais incapable de fournir une mesure de confiance. Il faut donc trouver une autre technique.

Le calcul de la distance de Kullback-Leibler entre la distribution de n-grammes de binaire et celles du corpus (cette valeur est aussi appelée divergence de Kullback-Leibler ou bien entropie croisée de chaînes de Markov) donne la même classification que *Multinomial Naive Bayes* sur le compte des n-grammes, en fournissant en bonus une mesure de confiance. [4] Comme `scikit-learn` ne fournit pas les moyens de faire le calcul de la distance de Kullback-Leibler, l'outil `cpu_rec.py` n'utilise pas `scikit-learn` et ré-implémente les structures de données creuses et les calculs de statistiques (le résultat est plus rapide que l'appel à MultinomialNB dans `scikit-learn`, mais évidemment plus lent qu'une implémentation en C). [5]

Selon les architectures, il peut être intéressant que la signature statistique inclue les n-grammes pour n = 2, 3, 4, ... ou éventuellement d'autres statistiques. Pour certaines architectures, le corpus est de trop petite taille et seuls les bigrammes et les trigrammes sont en quantité suffisante pour que la distribution observée ait un sens.

La technique utilisée par `cpu_rec.py` pour détecter si la reconnaissance a bien fonctionné est de considérer qu'il y a échec si l'analyse par bigrammes et l'analyse par trigrammes ne classent pas le même CPU en première position. Dans ce cas, en affichant des messages de debug, on peut connaître la liste des CPU reconnus (par distance de Kullback-Leibler croissante) selon l'analyse par bigrammes et selon l'analyse par trigrammes, afin d'avoir des suggestions de quoi regarder plus en détail. [6]

---

4. Le fait que cela fournisse la même classification a été observé expérimentalement ; en théorie, cela n'est pas toujours le cas, en particulier s'il y a de grandes différences de taille de corpus entre architectures.

5. Sur mon MacBook, `cpu_rec.py` met 25 secondes et prend 1 Go de RAM pour créer les signatures statistiques de 70 architectures. Avec `scikit-learn`, les statistiques sont calculés en 90 secondes et un peu plus de 1 Go. Quant à la version C de `cpu_rec.py`, elle n'utilise pas de représentation creuse pour les trigrammes, donc prend 8 Go de RAM pour 70 architectures, mais crée les signatures en 10 secondes.
Le temps d'analyse par `cpu_rec.py` est environ 60 secondes par Mo de fichier.

6. Ceci est en général inutile. Lors du développement de `cpu_rec.py`, ceci a surtout permisde détecter les cas où l'apprentissage d'un CPU à partir d'un corpus de taille insuffisante perturbe les résultats : ce nouveau CPU apparaît trop fréquemment en

La notion d'architecture reconnue par `cpu_rec.py` est donc une conséquence de la façon dont fonctionne cet outil. La classification obtenue dépend des propriétés statistiques des codes analysés.

En théorie, le compilateur utilisé a une influence sur les fréquences de bigrammes et trigrammes. Cette influence est souvent faible ; par exemple l'apprentissage de l'Intel x86 sur du code fabriqué par gcc ne complique pas la détection de code fabriqué par Visual Studio ou par Clang-LLVM. En revanche, le code 6502 fabriqué par `https://github.com/cc65/cc65` est caractéristique du compilateur plus que du CPU.

Certains CPU sont reconnus comme des architectures différentes, par exemple, le code x86 64-bit (AMD64) est différent de sa version 32-bit à cause de la présence de nombreux préfixes REX ; en revanche, le code PowerPC ou le code SPARC présente des statistiques similaires en version 32-bit ou 64-bit et l'outil ne fait pas la différence entre ces deux variantes d'un même jeu d'instructions. La reconnaissance d'architecture peut aussi être perturbée par l'utilisation massive d'instructions SSE/AltiVec/VIS/-Thumb, car les propriétés statistiques de ces instructions peuvent les faire considérer comme des architectures distinctes ; c'est le cas pour les binaires ARM hard float de Debian (architecture nommée `ARMhf` par l'outil), mais dans la plupart des cas il reste souvent suffisamment d'instructions "de base" pour que `cpu_rec.py` n'ait pas besoin de reconnaître une nouvelle architecture.

2.2 Fenêtres glissantes

En l'absence de conteneur (PE, ELF, Mach-O, COFF), il n'y a pas d'information indiquant où se situe le code exécutable donc `cpu_rec.py` doit détecter où est le code et où est le reste (sections de données, de relocation, de debug, ...). Il n'est pas rare qu'il y ait un peu de données au milieu du code exécutable (e.g. tables de switch) ; si ces données sont petites, `cpu_rec.py` ne les détectera pas, si elles sont longues, il détectera plusieurs sections de code exécutable.

L'analyse est faite avec une fenêtre glissante, suffisamment petite pour ne contenir que du code (afin que l'architecture soit reconnue sans que les statistiques ne soient perturbées par du non-code) et suffisamment grande pour que les calculs de fréquences dans la fenêtre se rapproche de la signature statistique de l'architecture. Une taille de fenêtre de 0x1000 octets est (heuristiquement) un bon ordre de grandeur ; quand le fichier

---

début de liste des CPU reconnus. L'affichage des informations détaillées sert aussi parfois lorsque le fichier à analyser ne fait pas plus de 200 octets.

à analyser est petit, l'outil diminue la taille de la fenêtre (jusqu'à 0x80 octets), avec moins de fiabilité dans la détection.

Une heuristique supplémentaire est utilisée : l'une des architectures apprises est le bytecode OCaml ; la signature statistique du bytecode OCaml n'est pas éloignée de celle des sections non exécutables dans les conteneurs habituels (PE, ELF, ...) et la distance de Kullback-Leibler entre les statistiques de bigrammes du bytecode OCaml du corpus et les statistiques de bigrammes d'un autre bytecode OCaml est très courte. Donc si l'architecture reconnue est OCaml, mais avec une distance de bigrammes un peu élevée, alors l'outil considère qu'aucune architecture n'a été reconnue.

3 À partir de quoi : création du corpus

Heuristiquement, un binaire avec une centaine de Ko d'instructions est suffisant pour avoir des statistiques permettant de reconnaître une architecture. Plus de 500 Ko est encore mieux, en particulier pour les trigrammes.

Pour les architectures usuelles, il existe des binaires librement téléchargeables, par exemple des versions de busybox ou C-Kermit ou des morceaux de distribution Linux. Cela permet de gérer les architectures les plus fréquentes (x86, ARM, MIPS, ...), et, grâce à C-Kermit, un grand nombre d'architectures antiques (m88k, we32k, Cray, ...)

Pour certaines architectures, par exemple V850, il est plus difficile de trouver où télécharger un code binaire pour l'apprentissage. Si cette architecture est connue d'un cross-compilateur (tel que gcc), alors il est possible de créer le corpus par cross-compilation d'un logiciel open source (zlib ou libjpeg, par exemple). Le plus compliqué pour la création du corpus par cross-compilation, ce sont les microprocesseurs 8-bits ayant une RAM et/ou ROM limitée, car au lieu de fabriquer un gros binaire il est nécessaire de fabriquer un grand nombre de petits fichiers. Dans le corpus fourni, j'ai parfois simulé ce grand nombre en répétant quelques petits fichiers.

Au résultat, la version actuelle de l'outil, avec son corpus par défaut, réuni ou fabriqué comme expliqué ci-dessus, reconnaît 72 architectures : 68HC08, 68HC11, 8051, ARM64, ARMeb, ARMel, ARMhf, ARcompact, AVR, Alpha, AxisCris, Blackfin, CLIPPER, Cell-SPU, CompactRISC, Cray, Epiphany, FR-V, FR30, FT32, H8-300, HP-Focus, HP-PA, IA-64, IQ2000, M32C, M32R, M68k, M88k, MCore, MIPS16, MIPSeb, MIPSel, MMIX, MN10300, MSP430, Mico32, MicroBlaze, Moxie, NDS32, NIOS-II, OCaml, PDP-11, PIC10, PIC16, PIC18, PIC24, PPCeb, PPCel, RISC-

V, RL78, ROMP, RX, S-390, SPARC, STM8, Stormy16, SuperH, TILE-Pro, TLCS-90, TMS320C2x, TMS320C6x, V850, VAX, Visium, WE32000, X86, X86-64, Xtensa, Z80, i860, et 6502 compilé avec `cc65`.

Plus de détails sur la constitution du corpus en annexe B.

4 Pour quoi : un module pour binwalk

Vu que `http://binwalk.org/` est l'outil classique pour l'analyse de firmware inconnu, l'intégration de cette technique de reconnaissance de CPU dans binwalk serait utile. L'architecture de binwalk est sous la forme de modules, chaque module pouvant être étendu par des plug-ins, mais aucun module actuel n'est adapté à l'ajout d'une méthode statistique de reconnaissance de CPU. J'ai donc rajouté à binwalk la possibilité de modules définis par l'utilisateur, cf. `https://github.com/devttys0/binwalk/pull/241`.

Avec un binwalk récent, il suffit donc de recopier `cpu_rec.py` à l'emplacement `$HOME/.config/binwalk/modules`, d'y extraire le corpus fourni `cpu_rec_corpus.tar`, et d'appeler binwalk avec l'option `-%`. Le corpus étant comprimé avec `xz`, il faut avoir installé le paquet `lzma` ou bien décomprimer le corpus après l'avoir extrait.

Le résultat ressemblera à

```
shell_prompt> binwalk -% corpus/PE/PPC/NTDLL.DLL corpus/MSP430/goodfet32.hex

Target File:   .../corpus/PE/PPC/NTDLL.DLL
MD5 Checksum:  d006a2a87a3596c744c5573aece81d77

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             None (size=0x5800)
22528         0x5800          PPCel (size=0x4c800)
335872        0x52000         None (size=0x1000)
339968        0x53000         IA-64 (size=0x800)
342016        0x53800         None (size=0x21800)

Target File:   .../corpus/MSP430/goodfet32.hex
MD5 Checksum:  4b295284024e2b6a6257b720a7168b92

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             MSP430 (size=0x5200)
20992         0x5200          None (size=0xe00)
```

Si la détection d'architecture échoue, le résultat ressemble à

```
shell_prompt> binwalk -% unknown/140

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             None (size=0xb400)
46080         0xB400          MMIX (size=0x80)
46208         0xB480          NDS32 (size=0x80)
46336         0xB500          None (size=0x14a80)
```

On peut remarquer que lors de l'analyse de `PPC/NTDLL.DLL`, un petit morceau de fichier a été détecté comme IA-64. Parmi les CPU étudiés,

l'architecture IA-64 a des distributions de bigrammes et trigrammes assez proches de distributions observées dans des sections de données. Cela perturbe la reconnaissance des architectures, et une piste serait de gérer cette architecture comme un cas particulier, en calculant une statistique supplémentaire. Par exemple, une statistique assez caractéristique du code IA-64 s'obtient en observant la distribution des séquences de deux instructions, ces instructions étant identifiées par leur premier octet. Il s'agit donc de compter les bigrammes du sous-ensemble du code binaire composé des octets d'adresse multiple de quatre. Néanmoins, en l'état actuel de l'outil, les résultats sont utilisables et donc ne nécessitent pas de gestion spécifique de l'architecture IA-64.

L'outil peut aussi être utilisé de façon autonome, indépendamment de binwalk. Sur la même entrée, sa sortie est un peu différente : en plus de l'analyse par fenêtre glissante, il y a le résultat d'une analyse de l'ensemble du fichier, et (parfois) une analyse de l'ensemble de la section de code exécutable. On peut ajouter une ou deux fois l'option `-v` pour augmenter la verbosité de l'outil et en particulier afficher les autres suggestions que la suggestion préférée. Cela sert par exemple à qualifier le corpus.

```
shell_prompt> cpu_rec.py corpus/PE/PPC/NTDLL.DLL corpus/MSP430/goodfet32.hex
corpus/PE/PPC/NTDLL.DLL       full(0x75b10)None    text(0x58800)PPCel chunk(0x4c800;153)PPCel
corpus/MSP430/goodfet32.hex   full(0x61ac) None                      chunk(0x5200;41)   MSP430
```

Le rajout de nouvelles architectures peut se faire en rajoutant des fichiers dans `$HOME/.config/binwalk/modules/cpu_rec_corpus`, ou bien en modifiant la méthode `read_corpus` dans `cpu_rec.py`.

5 Conclusion et perspectives

Cet outil `cpu_rec.py` met en œuvre une reconnaissance d'architecture efficace en pratique, et indiquant grossièrement où se trouvent les sections de code exécutable. C'est donc une nouvelle aide pour l'analyse de fichiers binaires, sous la forme d'une extension pour binwalk.

Il ne s'agit pas de repérer précisément l'emplacement des instructions dans le binaire, mais de détecter les cas où l'architecture présente est inhabituelle, exotique.

L'outil est disponible sous licence Apache 2.0 sur `https://github.com/airbus-seclab/cpu_rec`.

Une amélioration souhaitable est l'ajout de nouvelles architectures. Chaque utilisateur de l'outil peut le faire lui-même dans son corpus, mais il est recommandé de partager cette information afin qu'elle soit rajoutée dans le corpus libre disponible sur `https://github.com/airbus-seclab/cpu_rec/tree/master/cpu_rec_corpus`.

Avec cet outil, la trousse à outils n'est pas complète, et d'autres outils pourraient être utiles, a priori spécialisés pour certaines architectures :

— Un outil qui indique à quel(s) octet(s) commence le code exécutable. Les heuristiques d'IDA, en l'absence de spécification de l'*entry point* ou de symboles de fonctions, ne sont pas toujours fiables (essentiellement du désassemblage linéaire jusqu'à arriver à une instruction invalide, et une reconnaissance de motifs de prologues ou fins de fonctions). Il est donc parfois nécessaire de procéder par essais et erreurs. [7]

— Un outil qui retrouve l'adresse en mémoire à laquelle sont chargées les sections exécutables ; dans le cas d'un code PIE, cette question n'a pas de sens, mais sinon cette adresse devrait pouvoir être retrouvée en utilisant des particularités du jeu d'instructions.

— Un outil dédié ARM. L'architecture ARM est fragmentée en de nombreuses variantes, de plus en plus utilisée dans l'embarqué, et il n'est pas rare qu'un binaire contienne un mélange. `cpu_rec.py` reconnaît qu'il s'agit d'ARM, mais son approche statistique ne permettra pas plus de précision sur la nature et la localisation des divers morceaux de code exécutable.

6 Remerciements

Tout d'abord Raphaël Rigo, pour avoir exprimé le besoin d'un tel outil, et pour avoir commencé à réunir des éléments d'un corpus de binaires pour une variété de microprocesseurs, avec lequel il m'a été possible d'expérimenter avant de fabriquer moi-même un complément de corpus.

Et aussi divers interlocuteurs qui m'ont fait comprendre que mon idée de départ (proximité de chaînes de Markov) n'est pas à proprement parler une méthode de *machine learning*, m'ont orienté vers scikit-learn pour que je puisse expérimenter avec du *machine learning*, puis m'ont mentionné la notion d'*entropie croisée* qui est celle que je cherchais et que cet outil utilise : par ordre alphabétique Guillaume Charpiat, Vincent Feuillard, Pierre Senellart, Mehdi Tibouchi. Merci aussi à mes relecteurs : Philippe Biondi, Anaïs Gantet et Raphaël Rigo.

7. L'une des techniques expérimentées dans le cadre de la semaine REDOCS2016 utilisait Capstone Engine pour désassembler et en déduire si le CPU a été reconnu à partir de métriques calculées sur le résultat du désassemblage ; cette technique pourrait être adaptée pour détecter les points d'entrée, mais est limitée par le petit nombre d'architectures connues de Capstone Engine.

A Autres méthodes de reconnaissances d'architecture inconnues

Lors de la conception de `cpu_rec.py`, j'ai expérimenté avec les quelques autres méthodes ci-dessous, qui sont moins efficaces en pratique que la méthode retenue.

Autres techniques d'apprentissage. Au lieu d'utiliser directement la technique *Multinomial Naive Bayes*, il est habituellement recommandé de compenser les différences de taille de corpus en utilisant par exemple une transformation TF-IDF. En pratique cela a beaucoup diminué la capacité de reconnaissance de l'outil, et la technique utilisée (répétition manuelle des entrées pour les architectures ayant un trop petit corpus) permet d'éviter la plupart des effets nocifs au moment de l'ajout d'une nouvelle architecture.

D'autres classifieurs ont été testés et n'ont pas donné de meilleurs résultats. Néanmoins, il est probable que parmi les nombreux outils de classification existants, certains seront plus performant que celui utilisé par `cpu_rec.py`. Et si le corpus sur lequel apprendre les architectures exotiques était plus large, ces méthodes pourraient devenir intéressantes.

Statistiques modulo 4. Les architectures RISC 32-bits ont toutes leurs instructions de longueur 4 octets et alignées sur des adresses multiples de 4. Mais l'utilisation de 4-grammes, ou bien le calcul de statistiques différentes selon l'adresse modulo 4, ne donnent pas de meilleurs résultats que l'outil actuel, en particulier parce que cela demande que la fenêtre glissante soit plus longue que ce qui est permis par les statistiques de bigrammes et trigrammes.

Détection d'architecture RISC. Pour détecter si la longueur des instructions est n, on peut calculer la distribution des n sous-ensembles contenant les octets dont l'adresse a une valeur fixée modulo n. Si les distributions de ces n sous-ensembles sont suffisamment différentes, alors il est probable que la longueur des instructions soit un diviseur de n.

Cette approche marche assez bien pour détecter si une architecture est RISC 32-bits, pourvu qu'on sache isoler la section de texte. Donc en pratique cette technique n'apporte pas grand chose, d'autant plus qu'elle ne permet pas de savoir plus précisément quelle est l'architecture.

Recherche de motifs atypiques. Certaines séquences de 3 ou 4 octets, ou certains motifs spécifiques, sont caractéristiques d'une architecture. En

général, ce sont des séquences de prologue ou de fin de fonction. De telles séquences se trouvent par exemple au moyen d'une analyse statistique du corpus. En voici quelques unes, qui permettent de reconnaître certaines architectures avec grande fiabilité (même si les motifs IA-64 engendrent des faux positifs en particulier dans les sections de données, puisqu'ils ont deux octets nuls) :

`55 89 e5`	X86	`push %ebp; mov %esp,%ebp`
`55 57 56`	X86	`push %ebp; push %edi; push %esi`
`41 57 41 56`	X86-64	`push %r15; push %r14`
`41 55 41 54`	X86-64	`push %r13; push %r12`
`55 48 89 e5`	X86-64	`push %rbp; mov %rsp,%rbp`
`0e f0 a0 e1`	ARMel	`ret` (typique de gcc 4.x)
`1e ff 2f e1`	ARMel	`bx lr` (typique de gcc 3.x)
`6b c2 3f d9`	HP-PA	`stw %rp, -cur_rp(%sp)`
`4e 5e 4e 75`	M68k	`unlk a6; rts`
`ff bd 67`	MIPSel	`daddiu $sp, -X`
`ff bd 27`	MIPSel	`addiu $sp, -X`
`67 bd ff`	MIPSeb	`diaddu $sp, -X`
`67 bd 00`	MIPSeb	`diaddu $sp, +X`
`03 99 e0 21`	MIPSeb	`addu $gp, $t9`
`4e 80 00 20`	PPCeb	`blr`
`81 c3 e0 08`	sparc	`retl`
`60 00 80 00`	IA-64	`br.few b6`
`08 00 84 00`	IA-64	`br.ret.sptk.many b0`

Cette approche permet d'avoir une réponse plus vite que `cpu_rec.py`, mais elle est moins générale : le calcul de distances entre distributions de trigrammes permet d'utiliser automatiquement le fait que sur x86 la séquence `55 89 e5` est bien plus probable que pour les autres architectures, mais si le compilateur utilisé ne produit pas les instructions `push %ebp; mov %esp,%ebp`, alors les autres motifs fréquents seront automatiquement pris en compte.

Ceci se voit par exemple pour la reconnaissance de l'ARMel (little-endian) pour laquelle le corpus de `cpu_rec.py` est construit en utilisant uniquement un binaire compilé avec gcc 3.x, mais permet de reconnaître les binaires compilés avec gcc 4.x, bien que la plupart d'entre eux ne contienne aucune instruction `bx lr`.

B Plus de détails sur la constitution du corpus

Le corpus est un élément essentiel de `cpu_rec.py` : l'outil a été conçu pour pouvoir apprendre chance nouvelle architecture avec seulement quelques centaines de Ko, mais l'intérêt de l'outil réside en sa capacité à reconnaître des architectures inhabituelles.

C'est pour cela que la constitution du corpus est détaillée, et que des extensions de ce corpus sont bienvenues.

B.1 Binaires divers

Pour diverses architectures, le corpus se base sur quelques un des fichiers ELF fournis par Raphaël Rigo (un fichier par architecture, c'est suffisant) ; l'apprentissage est fait sur la section de code exécutable [8]. Ce sont des versions de `libgmp.so`, `libm.so` ou `libc.so` issues de diverses distributions Debian (pour x86, x86_64, m68k, PowerPC, S/390, SPARC, Alpha, HP-PA, MIPS et quelques variantes de ARM ; le code source correspondant à ces binaires est sur `http://archive.debian.org/`) ou bien un busybox (pour ARM big endian et SH-4, binaires disponibles sur `https://busybox.net/downloads/binaries/`).

Pour d'autres architectures, de nombreux binaires sont disponibles sur `ftp://kermit.columbia.edu/kermit/bin/` : cela a permis d'enrichir le corpus avec M88k, HP-Focus, Cray, Vax, PDP-11, ROMP, WE32k, CLIPPER, i860. Certains de ces fichiers sont au format COFF, qui indique où est la section .text, pour d'autres il a fallu utiliser l'outil `cpu_rec.py` en mode verbeux pour en déduire où est le code exécutable. Le corpus utilise aussi un firmware pour TMS320C2x fourni par Raphaël Rigo, au format COFF, issu de `https://github.com/slavaprokopiy/Mini-TMS320C28346/blob/master/For_user/C28346_Load_Program_to_Flash/Debug/C28346_Load_Program_to_Flash.out`.

Pour presque toutes ces architectures, un unique binaire a été utilisé pour faire partie du corpus. Cela peut paraître audacieux de ne pas utiliser de nombreux binaires variés, quand ils sont disponibles, mais en pratique cela n'est en général pas nécessaire, et cela permet de valider que l'approche statistique utilisée par `cpu_rec.py` sera valide même dans les cas où un seul binaire a pu être trouvé. Un exemple est l'architecture

8. La section de code exécutable s'appelle normalement `.text` en COFF, PE et ELF, et `__TEXT,__text` en Mach-O. Ces noms sont des conventions, un exécutable valide pourrait utiliser d'autres noms. Les binaires utilisés pour le corpus respectent cette convention.

CLIPPER, apprise à partir d'un unique binaire (celui de C-Kermit), et détectée dans les fichiers `boot.1` de `https://web-docs.gsi.de/~kraemer/COLLECTION/INTERGRAPH/starfish.osfn.org/Intergraph/index.html`.

B.2 Cross-compilateur pour les architectures connues de `gcc.gnu.org`

La première étape est d'installer les binutils et le compilateur. Les instructions ci-dessous ont fonctionné sous MacOSX, et devraient aussi fonctionner aussi sous Linux (à quelques modifications de path près).

Les architectures connues du gcc de `gcc.gnu.org` sont : `aarch64` `alpha` `arc` `arm` `avr` `bfin` `c6x` `cr16` `cris` `epiphany` `fr30` `frv` `ft32` `h8300` `i386` `ia64` `iq2000` `lm32` `m32c` `m32r` `m68k` `mcore` `microblaze` `mips` `mmix` `mn10300` `moxie` `msp430` `nds32be` `nds32le` `nios2` `nvptx` `pa` `pdp11` `rl78` `rs6000` `rx` `s390` `sh` `sparc` `spu` `tilegx` `tilepro` `v850` `visium` `xstormy16` `xtensa` mais la création d'un cross-compilateur avec la recette ci-dessous échoue pour quelques-unes [9].

```
export PREFIX=$BASEDIR/cross
export PATH="$PREFIX/bin:$PATH"
mkdir -p $BASEDIR/corpus $PREFIX
cd $BASEDIR
git clone git://sourceware.org/git/binutils-gdb.git
svn checkout svn://gcc.gnu.org/svn/gcc/trunk gcc

# the list of known architectures is in gcc/gcc/config
export TARGET=v850-elf

mkdir $BASEDIR/build-binutils-$TARGET
cd $BASEDIR/build-binutils-$TARGET
../binutils-gdb/configure --target=$TARGET --prefix="$PREFIX" \
    --with-sysroot --disable-nls --disable-werror
make
make install

mkdir $BASEDIR/build-gcc-$TARGET
cd $BASEDIR/build-gcc-$TARGET
../gcc/configure --target=$TARGET --prefix="$PREFIX" \
    --disable-nls --enable-languages=c --without-headers \
    --with-libiconv-prefix=/usr --with-gmp=/opt/local
make all-gcc
make all-target-libgcc
```

9. `mmix` : il faut utiliser `TARGET=mmix` et non pas `TARGET=mmix-elf` ;
`pdp11` : bug au moment de la création de l'assembleur ;
`tilegx` : bug pour la création de gcc (fichier `tilepro/gen-mul-tables.cc` invalide).

```
make install-gcc
make install-target-libgcc
```

Les binutils et gcc ne suffisent pas pour compiler la zlib par exemple, car il manque en particulier la libc. Seuls des programmes sans aucune dépendance sont compilables.

L'approche habituelle est d'installer d'autres éléments, selon l'architecture visée, par exemple la libgloss ou la libnosys (`https://github.com/32bitmicro/newlib-nano-1.0/` est un bon point d'entrée). Parfois l'absence de FPU demande qu'existent des fonctions telles que `__divsf3`, qui selon les architectures seront dans la libm ou la libgcc.

Au lieu de suivre cette approche, j'ai créé des stubs vides, ce qui a permis de directement gérer des architectures non connues de la newlib, telles que H8/300 ou FTDI FT32.

B.3 Cross-compilation de zlib et libjpeg

Ce sont des bibliothèques plutôt calculatoires, donc avec peu d'adhérence au système d'exploitation, ce qui en fait de bonnes candidates pour une création de corpus d'instructions d'un CPU.

Les lignes de commande pour une cross-compilation dépendent un peu de la bibliothèque à compiler. On utilise pour le corpus les exécutables produits : minigzip et jpegtran. Par exemple pour V850 cela donne :

```
TARGET=v850-elf

ZLIB=zlib-1.2.10
curl -O http://zlib.net/$ZLIB.tar.gz
tar xzf $ZLIB.tar.gz
cd $ZLIB
CROSS_PREFIX=$TARGET- uname=cross ./configure
make clean
make CC="$TARGET-gcc $CFLAGS"
cp minigzip .../minigzip-$TARGET

curl -O http://www.ijg.org/files/jpegsr6b.zip
unzip -x jpegsr6b.zip
cd jpeg-6b
perl -pi -e 's/\r$//' ./configure
CC="$TARGET-gcc $CFLAGS" ./configure
make clean
make AR="$TARGET-ar rc" AR2="$TARGET-ranlib"
cp jpegtran .../jpegtran-$TARGET
```

Dans plusieurs cas (c6x, cr16, epiphany, rl78, tilepro) en suivant la procédure ci-dessus, on tombe sur des bugs du cross-compilateur gcc (*internal error* ou *SEGV* – les bug reports restent à faire) qui peuvent être contournés en modifiant le source de zlib ou libjpeg. De plus, les exécutables cross-compilés pour RL78 ou MSP430 (avec `-mlarge`) ont une section .text inutilisable et le corpus ne peut se baser sur minigzip et jpegtran. À la place, il se base sur les sections .text des fichiers objet engendrés.

B.4 Autres cross-compilateurs

Il existe quelques cross-compilateurs basés sur gcc mais disponibles ailleurs que sur `gcc.gnu.org`, par exemple la toolchain gcc/newlib disponible sur `https://riscv.org/software-tools/`, dont un binaire a rejoint le corpus pour l'architecture RISC-V.

Mais pour de nombreux microprocesseurs 8-bits, non seulement le gcc de `gcc.gnu.org` ne sait pas faire de cross-compilation, mais comme ces microprocesseurs ont un espace mémoire limité, on ne peut y faire tenir la zlib ou la libjpeg. Il faut donc procéder autrement. Au lieu de compiler une bibliothèque entière, on se limite à de petits programmes. Pour construire le corpus, principalement trois ont été utilisés ; ça n'est pas suffisant pour avoir une détection de CPU de bonne qualité, mais cela suffit à prouver la validité de l'approche.

Pour le MC68HC11 il y a un cross-compilateur fourni avec Ubuntu 12.04, par exemple, qui fabrique des binaires ELF. Le compilateur disponible sur `http://sdcc.sourceforge.net/` fabrique des fichiers au format HEX, pour plusieurs variantes de 8051 (mcd51, ds390, ds400), de Z80 (z80, z180, r2k, r3ka [10]), pour STM8, et au format ELF pour des MC68HC08 (hc08, s08).

Pour le 6502, `https://github.com/cc65/cc65` permet de fabriquer du code, mais celui-ci est trop caractéristique du compilateur, plutôt que du microprocesseur (cela se voit en regardant la régularité de l'assembleur engendré, et en pratique l'apprentissage sur ce code ne permet par exemple pas de reconnaître des ROM Apple II). Le corpus par défaut ne permet donc pas de reconnaître le 6502 mais uniquement une variante que je nomme `#6502#cc65`.

10. La compilation avec `-mtlcs90` n'est pas détectée comme fabriquant du code Z80, ce qui est surprenant car (contrairement au TLCS 900) le TLCS 90 est binairement compatible avec le Z80. Le corpus les considère donc comme deux architectures distinctes.

B.5 En l'absence de cross-compilateur

Il n'y a pas de compilateur libre permettant de fabriquer du code PIC [11], Il faut donc trouver des binaires pour les nombreuses variantes de PIC (principalement PIC10, PIC16, PIC18, PIC24). Le corpus inclut un firmware pour PIC18 (issu de `https://github.com/radare/radare2-regressions/blob/master/bins/pic18c/FreeRTOS-pic18c.hex`) et un autre pour PIC24 (issu de `https://raw.githubusercontent.com/mikebdp2/Bus_Pirate/master/package_latest/BPv4/firmware/bpv4_fw7.0_opt0_18092016.hex`) et des petits firmwares pour PIC10 et PIC16 (issus de `http://www.pic24.ru/doku.php/en/osa/ref/examples/intro`).

B.6 Qualité du corpus obtenu

Le corpus d'apprentissage doit être discriminant : deux labels différents doivent correspondre à deux comportements statistiques différents. Pour satisfaire ce critère, les labels ont été définis progressivement. Par exemple, lorsque l'architecture SPARC a été apprise sur des binaires SPARC v7, l'outil a été utilisé pour analyser des binaires SPARC v9 (64-bits, et avec un jeu d'instructions ayant des extensions) : la plupart de ces binaires ont été reconnus comme SPARC, donc la conclusion est que ces architectures sont proches ; ensuite l'outil a été utilisé sur des binaires SPARC v7 et SPARC v9, avec un apprentissage sur un corpus différenciant v7 et v9 : les binaires v7 ont été reconnus comme v7 ou v9, et les binaires v9 ont été reconnus comme v7 ou v9, donc la conclusion est que l'approche par bigrammes et trigrammes ne discrimine pas entre ces deux architectures ; le corpus ne contient donc qu'une architecture SPARC.

Pour une architecture donnée, le corpus doit être suffisant : il faut suffisamment de données pour que le comptage des bigrammes et trigrammes fasse émerger des caractéristiques de cette architecture. Lorsqu'un corpus pour une architecture est insuffisant, l'outil se met à fournir des réponses erronées pour les autres architectures : le corpus insuffisant a une distribution trop peu marquée qui perturbe les calculs de proximité. Une solution (en l'absence de données supplémentaires) est de relire de façon répétée les données au moment de l'apprentissage (c'est une solution parce que le calcul de la distance de Kullback-Leibler, afin d'éviter des divisions par 0, additionne une distribution uniforme à la distribution observée dans le corpus ; répéter le corpus revient à diminuer le poids de cette distribution uniforme).

L'amélioration du corpus peut se faire dans deux directions :

11. SDCC a des options -mpic16 et -mpic14, mais elles ne sont pas fonctionnelles.

— Extension du corpus pour les architectures pour lesquelles celui-ci est insuffisant. Si on calcule la taille (comprimée par xz) de chaque fichier du corpus, les plus petits sont ceux pour lesquels le corpus contient le moins d'information. On en déduit que les CPUs pour lesquels les données sont le plus incomplètes sont : PIC10, STM8, PIC16, PIC18, TMS320C2x, puis Z80, 8051, 68HC08, 68HC11 et TLCS-90.
— Rajout de nouvelles architectures. Voici une liste de quelques architectures qui ne sont pas présentes dans le corpus, et pour lesquelles il faudrait vérifier si elles ne sont pas proches d'une architecture connue, ou bien les rajouter dans le corpus[12] : 6502, 6800 (si différent du PDP-11 et des 68HCx), 6809, 8080 (incl. 8085), AM29k, B5000, CDC-*, CEVA-XC, CEVA-X, CEVA-Teaklite, DSP56k, Elbrus VLIW, eSi-RISC, F8, F18A, HD6301, KDF9, i960, MARC4, MCS-48, Mico8, MSC81xx, OpenRISC, PDP-1, PDP-7, PDP-8, PDP-10, PSC1000, Propeller, RTX2000, S1C6x, Saturn, SHARC, Signetics 2650, SPC, SystemZ (si différent du S/390), TMS320C1x, TMS320C3x, TMS320C5x, Transputer, TriMedia, xCore.

12. Cette liste d'architectures existantes est issue principalement de `https://en.wikipedia.org/wiki/Microprocessor`, `https://en.wikipedia.org/wiki/List_of_instruction_sets`, `https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures`, `https://en.wikipedia.org/wiki/Digital_signal_processor` et `https://github.com/larsbrinkhoff/awesome-cpus`.