Et TINA RUSTINA le lien vers l'assembleur

Frederic Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, Marie-Laure Potet

Et TINA RUSTINA le lien vers l'assembleur
– étude préliminaire –

Frédéric Recoules[1], Sébastien Bardin[1], Richard Bonichon[2],
Laurent Mounier[3] et Marie-Laure Potet[3]

[1] CEA LIST, Laboratoire de Sûreté et Sécurité du Logiciel, Paris-Saclay, France
prenom.nom@cea.fr
[2] Tweag I/O, Paris, France
prenom.nom@tweag.io
[3] Univ. Grenoble Alpes. VERIMAG, Grenoble, France
prenom.nom@univ-grenoble-alpes.fr

Résumé

Le développement en C de logiciels non-critiques utilise régulièrement l'insertion d'assembleur « en ligne », que ce soit pour optimiser certaines opérations ou pour accéder à des primitives systèmes autrement inaccessibles. Celui-ci demande l'écriture de spécifications sous forme de contraintes pour faire le lien entre le langage hôte (C) et le langage assembleur embarqué. Ces spécifications sont ensuite utilisées par le compilateur afin d'insérer aveuglément l'assembleur dans le code émis – la pratique étant d'accorder toute confiance aux dires du programmeur. Pour éviter les erreurs issues de spécifications mal formées, nous proposons RUSTINA, un outil qui permet de vérifier que les spécifications assembleurs correspondent bien à l'implémentation des blocs qu'elles décrivent, ou, dans le cas contraire diagnostiquer ou corriger le problème.

1 Introduction

Contexte. Nous considérons ici la programmation de code « mixte », combinant assembleur embarqué et code C/C++. Cette fonctionnalité, présente dans les compilateurs GCC, clang et icc (Intel), permet d'intégrer des instructions assembleur au sein d'un programme C/C++. Elle est utilisée en général pour des raisons d'efficacité ou d'accès à des fonctionnalités bas niveau qui ne peuvent être déclenchées depuis le langage hôte. L'usage d'assembleur en ligne est plus fréquent que l'on n'imagine : ainsi **11%** des paquets Debian 8.11 écrits en C/C++ utilisent de l'assembleur en ligne, directement ou à travers des bibliothèques, avec des blocs allant jusqu'à 500 instructions, et **28%** des projets C les plus populaires sur Github en contiennent, d'après Rigger et coll. [8]. L'assembleur en ligne est ainsi fréquemment utilisé dans des domaines comme la cryptographie, le multimédia ou les pilotes matériel.

Problème. L'écriture d'assembleur en ligne comporte par nature un risque important : le compilateur fait entièrement confiance à l'interface du bloc assembleur, et notamment aux spécifications qu'elle comporte. Ceci peut donc entraîner des problèmes, notamment lorsque les spécifications reflètent incorrectement la sémantique du bloc concerné, ou réduire les opportunités d'optimisations – et avec elle les performances pourtant visées – si les contraintes se révèlent trop fortes.

Objectif. Nous souhaitons concevoir et développer une technique automatique, générique et pratique pour s'assurer de la conformité des instructions assembleurs avec les spécifications de l'interface qui leur correspond. De manière générale, nous entendons ainsi :

— vérifier la correction et la complétude de l'interface par rapport au code qu'elle décrit ;

— proposer un correctif pour que ces propriétés soient respectées si ce n'est pas le cas.

Fehnker et coll. [4] ont abordé ce problème de vérification des spécifications de l'interface. Cependant, leur technique ne couvre pas l'ensemble des propriétés que nous jugeons indispensables. Par ailleurs, la technique requiert un analyseur syntaxique dédié à l'assembleur embarqué pour chaque architecture, mais aussi, pour chaque dialecte assembleur. Testée sur ARM, elle souffre ici d'un manque de généricité et pourrait se révéler fragile sur une architecture de type *CISC*, plus complexe.

Contribution. Nous proposons RUSTINA, Rafistolage Utile de Spécifications par TINA, un outil de révision d'interface pour extraire la sémantique du bloc assembleur, vérifier sa conformité et proposer une rustine corrective si la correction n'est pas assurée, mais aussi, si le bloc est sur-contraint. Nos contributions se situent à 2 niveaux :

— l'identification des propriétés de conformité attendues entre une interface et son code assembleur (Sec. 3),

— un algorithme de vérification desdites propriétés et de génération automatique de correctifs, implémenté au-dessus de TINA[6, 7] (Sec. 4) et testé sur des projets d'envergure (Sec. 5).

Discussion. Notre technique trouve et corrige des bugs de *non-conformité* au niveau des blocs assembleur embarqué. En fonction du contexte, certains produiront alors des erreurs à l'exécution (Sec. 2), là où d'autres resteront fortuitement silencieux (Sec. 5).

2 Contexte et motivation

```
1563  # ifdef __PIC__
1564
1565  __STRING_INLINE size_t
1566  __strcspn_g (const char *__s, const char *__reject)
1567  {
1568    register unsigned long int __d0, __d1, __d2;
1569    register const char *__res;
1570    __asm__ __volatile__
1571      ("pushl        %%ebx\n\t"
1572       "movl         %4,%%edi\n\t"
1573       "cld\n\t"
1574       "repne; scasb\n\t"
1575       "notl         %%ecx\n\t"
1576       "leal         -1(%%ecx),%%ebx\n"
1577       "1:\n\t"
1578       "lodsb\n\t"
1579       "testb         %%al,%%al\n\t"
1580       "je         2f\n\t"
1581       "movl         %4,%%edi\n\t"
1582       "movl         %%ebx,%%ecx\n\t"
1583       "repne; scasb\n\t"
1584       "jne         1b\n"
1585       "2:\n\t"
1586       "popl         %%ebx"
1587       : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588       : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589       : "memory", "cc");
1590    return (__res - 1) - __s;
1591  }
```

FIGURE 1 – Code assembleur en ligne issu de libc6-dev

Exemple. L'extrait de code assembleur en ligne (x86) en Fig. 1 provient du fichier `/bits/string.h` du paquet libc6-dev (2.19-18) et nous servira d'exemple. Cette fonction calcule la longueur de la sous-chaîne de `__s` qui ne contient aucun caractère de `__reject`. Regardons comment l'assembleur est lié au contexte C : les lignes 1587 à 1589 déclarent un contrat composé de 3 listes (séparées par `':'`) :

les sorties, les entrées et les effets bord (appelés *clobbers*) du bloc. Les entrées et sorties associent des variables ou des expressions du C à des registres ou emplacements mémoire de l'assembleur. Elles sont numérotées dans l'ordre d'apparition, à partir de `0`, ce qui permet d'y faire référence dans les mnémoniques – le compilateur remplacera lors de l'émission du code les `%0-9` par l'emplacement qu'il aura choisi en respectant les contraintes. Ces dernières se composent de classes génériques (`'m'` mémoire, `'r'` registre général, etc.), de classes spécifiques (`'a'` `eax`, `'A'` `edx`::`eax`, etc.) et de modificateurs (`'='` écriture, `'+'` lecture/écriture, etc.). Il est possible de partager explicitement un emplacement entre une sortie et une entrée, en attribuant le numéro de la sortie à l'entrée (`"0"`). Le compilateur peut également le faire de son propre chef – il cherchera à minimiser le nombre de registres alloués et, pour ce faire, travaille avec l'hypothèse que les entrées sont « consommées » avant que les sorties ne soient « produites ». Dans cette optique, il est alors valide de partager le même registre. Toutefois, si la structure du code ne respecte pas cette hypothèse, le modificateur `'&'`, « écriture en avance », permet de forcer le compilateur à choisir un registre de sortie distinct de toutes les entrées.

En détail, voici comment interpréter le contrat en Fig. 1 :

%0. `"=S" (__res)` indique que le registre `esi` retournera la nouvelle valeur de `__res` et `"0" (__s)` qu'il doit contenir la valeur de `__s` à l'entrée du bloc.

%1-2. `"=&a" (__d0)` (respectivement `"=&c" (__d1)`) indique que le registre `eax` (respectivement `ecx`) sera écrasé – `__d0` (respectivement `__d1`) n'étant pas utile au delà du bloc, il s'agit d'un registre intermédiaire (scratch register) – et `"1" (0)` (respectivement `"2" (0xffffffff)`) qu'il doit contenir la valeur `0` (respectivement `0xffffffff`) à l'entrée du bloc.

%3. `"=&D" (__d2)` indique, comme précédemment, que le registre `edi` est un registre intermédiaire. Sa valeur à l'entrée du bloc n'est pas définie.

%4. `"r" (__reject)` indique qu'un registre en lecture seule – au choix du compilateur – doit contenir la valeur de `__reject`.

⚠. `"memory"` et `"cc"` indiquent que le bloc peut accéder et modifier respectivement tout emplacement mémoire accessible depuis l'environnement et le registre de condition.

Nous pouvons déduire de ces informations quelles sont les entrées qui partagent ou ne peuvent partager un registre avec une sortie ; ces données sont illustrées dans la Fig. 2. Le compilateur a ici peu de marge de manœuvre, l'interface est très contrainte car les mnémoniques sont très spécialisées et utilisent implicitement des registres particuliers. Le seul choix qu'il lui reste à faire est d'*associer* un registre à l'entrée `%4` (lignes 1572 et 1581), en sachant qu'il ne peut choisir `eax`, `ecx` ou `edi` à cause du modificateur `'&'`. De plus, tant que les valeurs de `__s` et `__reject` ne sont pas démontrées égales, `%4` se doit d'être distinct d'`esi`. Ce choix se résume donc aux registres `edx` et `ebx` – il semble d'apparence simple mais nous allons expliquer maintenant pourquoi il peut être fatal.

Le fond du problème. En effet, si `ebx` n'apparaît pas dans le contrat, il est bien utilisé dans le code assembleur (écrit ligne 1576, lu ligne 1582), à l'insu des compilateurs, puisque ceux-ci n'inspectent pas les mnémoniques. Ainsi, si ce registre est choisi, la valeur de l'entrée (le pointeur `__reject`) sera écrasée à la ligne 1576 par le calcul d'`ecx` (la longueur de la chaîne pointée par `__reject`). Le problème est alors d'autant plus important que, s'agissant d'un pointeur, la lecture (ligne 1581) et le déréférencement (ligne 1583) de la mauvaise valeur provoquera une *erreur de segmentation*. Cette erreur est ainsi observable à l'exécution lorsque le code est compilé avec GCC 5.4 et les options `-m32 -O3 -fpic`.

Remarque. La forme du contrat observée ici est probablement due au fait que le compilateur refusait, au moins jusqu'à GCC 4.8, la déclaration d'`ebx` en *clobber* si l'option `-fpic` était active : il était ainsi utilisé de façon détournée, en prenant soin de le sauvegarder (`push` ligne 1571) et de le restaurer (`pop` ligne 1586). Toujours est-il que l'hypothèse *contextuelle* – `ebx` ne sera pas choisi – est particulièrement fragile avec l'évolution des compilateurs et la réutilisation de code. Aussi, afin de ne plus souffrir de

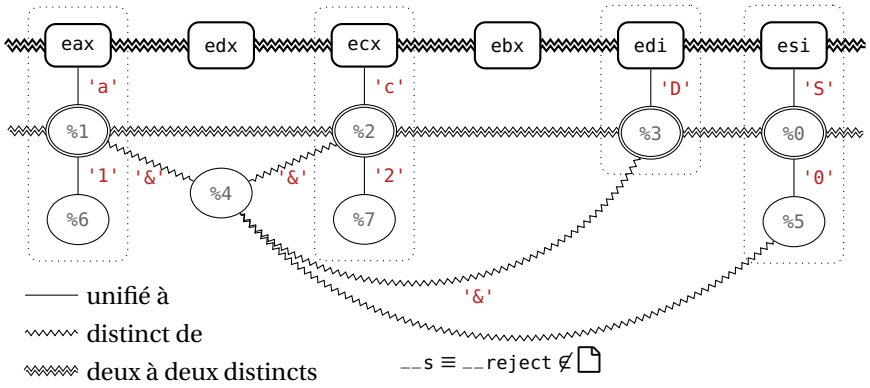

FIGURE 2 – Diagramme de juxtaposition

pareille ambiguïté, *il faut associer explicitement* `%4` au registre `edx` à l'aide de `'d'` au lieu de `'r'`, ce que l'on peut observer dans des versions plus récentes de libc6-dev. Cet extrait montre donc que de subtiles erreurs peuvent aussi se glisser dans le code de programmeurs expérimentés.

Conclusion. À la complexité d'écrire de l'assembleur s'ajoute ainsi celle du contrat le liant au C. C'est pourquoi nous proposons en Sec. 3 de définir plus en détail les propriétés attendues de l'assembleur embarqué avant de présenter les caractéristiques techniques de notre outil RUSTINA en Sec. 4. Les résultats préliminaires de notre prototype sont eux discutés en Sec. 5.

3 Détails des propriétés recherchées

Nous définissons un ensemble de 5 propriétés que se doit de respecter une interface vis-à-vis de ses mnémoniques assembleur pour être qualifiée de ***BUENO***. Les 4 premières ***BUEN*** sont des propriétés de correction : elles sont nécessaires au bon fonctionnement du code. Elles sont menacées par les sous-spécifications qui, à l'instar des comportements indéfinis, provoquent de subtils bogues selon le bon vouloir du compilateur. La dernière, ***O*** est une propriété de complétude et n'est, stricto sensu, pas nécessaire. Elle est menacée par les sur-spécifications qui risquent de réduire l'efficacité du programme, allant à l'encontre même de l'utilisation de l'assembleur. Détaillons un peu plus ce que recouvrent ces propriétés :

***B*ien-formée.** *Quelle que soit l'instanciation, les mnémoniques assembleur expansées sont valides.*

Il s'agit ici d'établir que les contraintes déclarées respectent celles de l'architecture cible, par exemple, qu'il n'est pas autorisé de choisir deux accès mémoires `"m"` pour un `mov` x86. Considérée comme moins importante, nous ne développons pas cette propriété dans ce papier – une erreur sera mise en exergue par le processus de compilation, sans conséquence donc à l'exécution.

***U*tilisable.** *Si une valeur de retour – en écriture seule – est utilisée par le C, elle est correctement initialisée par le bloc.*

Cette propriété permet d'assurer que toutes les valeurs de retour sont bien définies.

***E*xhaustive.** *Chaque emplacement – registre ou mémoire – sur lequel le bloc peut agir est correctement décrit dans l'interface.*

Cette propriété s'assure que l'interface est complète, c'est-à-dire que les emplacements lus sont préalablement initialisés ou déclarés en lecture et que les emplacements écrits sont soit des sorties,

soit listés comme *clobbers*. La section 2 illustre un des risques du non-respect de ***E***. L'autre risque provient des optimisations que le compilateur va réaliser en se basant uniquement sur ces déclarations, pouvant fausser les véritables chaînes de dépendances – l'affectation non déclarée d'une variable pourrait, par exemple, être ignorée si le bloc est jugé inutile à tort, ou si sa précédente valeur se voit propagée.

***N*on-ambigüe.** *Quelle que soit l'instanciation, le rôle de chaque opérande est le même.*

À l'instar des macros en C, des problèmes peuvent survenir lors du remplacement des opérandes par leur emplacement en présence d'« aliasing ». Un bloc est ambigu si le compilateur est libre de partager le même registre entre une entrée et une sortie malgré le fait que cette sortie est écrite avant que l'entrée ne soit lue. Deux productions sémantiques seront alors observables suivant qu'il décide ou non de faire ce partage. Une ambiguïté apparaît également si un même objet C est associé à différents registres de sortie. L'ordre de *considération* des sorties n'étant pas défini, la valeur reçue par cet objet dépendra ainsi du choix d'implémentation. L'exemple de la Fig. 3 illustre ce problème : les registres `eax` et `edx` sont tous les deux censés venir écraser la valeur de `x` – sans aucune forme d'avertissement, GCC choisira la valeur d'`edx` (`1`) alors qu'icc préférera celle d'`eax` (`0`).

```
int f () { int x; __asm__ ("" : "=a" (x), "=d" (x) : "0" (0), "1" (1)); return x; }
```

FIGURE 3 – Exemple minimal d'interface ambiguë

***O*ptimale.** *Le compilateur dispose de toutes les libertés pour produire le code le plus efficace.*

Cette propriété s'assure quant à elle que l'interface n'a pas été sur-spécifiée. Pour cela, il est nécessaire que chacun des éléments qui la compose soit utile et justifié et que l'application de modificateurs soit réduite au strict minimum requis par la correction. L'exemple le plus répandu de sur-spécification est l'usage du *clobber* `"memory"` là où des alternatives plus précises permettraient au compilateur de ne pas avoir à oublier toutes ses connaissances sur la mémoire.

4 Implémentation

Notre prototype RUSTINA repose sur l'extraction de la sémantique des mnémoniques proposée pour TINA [6, 7] – outil d'aide à la vérification de code mixte C et assembleur embarqué – à laquelle nous ajoutons une analyse dédiée pour établir des diagnostics automatiques et générer les correctifs correspondants. Les Tables 1 and 2 donnent les propriétés prises en charge par RUSTINA.

TABLE 1 – Aperçu des propriétés

	Garantit[1]	Réfute[1]	Rustine
B	hors-sujet		
U	✓	✓	✓
E	✓	~	~
N	✓	×	×
O	×	✓	✓

Contexte. Une première analyse de la fonction C à l'aide de Frama-C [5] permet de déterminer les variables ***U***tiles à la sortie du bloc – afin de distinguer les sorties des temporaires. L'interface du bloc assembleur est ensuite parcourue, les objets du C ainsi que les registres de la section *clobbers* sont étiquetés en fonction des spécifications (*l* – lecture, *e* – écriture). Un diagramme de juxtaposition est ensuite calculé (comme dans la Fig. 2). La production de ce diagramme nécessite par ailleurs une connaissance de l'architecture visée, plus précisément sur les classes de registres qu'elle possède et la façon dont sont peuplées les contraintes.

Extraction. Comme pour TINA, le code est ensuite compilé avec les informations de débogage DWARF [3] – le compilateur ne joue qu'un rôle minime, se contentant de transférer l'assembleur

1. *Garantie* : si l'outil ne rapporte pas d'erreur alors il n'y en a pas – *Réfuté* : les erreurs rapportées sont vraies.

TABLE 2 – Garanties, détections de bug et correctifs apportés par RUSTINA

	Règle	Garantit	Réfute	Rustine
U	Pas de sortie non initialisée	✓	✓	*R*0. promeut la sortie en lecture & écriture `'+'`[0]
E	Pas d'emplacement lu sans permission :			
	– registre libre	✓	✓	hors-sujet[1]
	– registre *clobber* avant écriture	✓	✓	rejet justifié
	– sortie	✓	✓	*R*0. promeut la sortie en lecture & écriture `'+'`[0]
	Pas d'emplacement écrit sans permission :			
	– registre libre	✓	✓	*R*1. ajoute le registre à la liste des *clobbers*
	– registre d'entrée	✓	✓	*R*2. promeut l'entrée en lecture & écriture `'+'`
	Pas d'accès mémoire non déclaré :			
	– via un pointeur d'entrée	✓	×	*R*3. ajoute le *clobber* `"memory"`
	– via un symbole global	✓	✓	*R*4. ajoute une nouvelle entrée de type `"m"`[2]
N	Pas de doublon de sorties par registre :			
	– variable locale	✓	✓	rejet justifié
	– contenu pointé	✓[3]	×	avertissement indicatif
	Pas de lecture d'entrée altérée	✓	×	rejet indicatif
O	Pas d'entrée non lue :			
	– lecture seule	✓	✓	*R*5. retire l'élément de la liste
	– lecture & écriture	✓	✓	*R*6. transforme en sortie en écriture seule
	Pas de sortie ni utilisée ni écrite	✓	✓	*R*5. retire l'élément de la liste
	Pas de registre *clobber* non écrit	✓	✓	*R*5. retire l'élément de la liste
	Pas de *clobber* `"memory"` superflu :			
	– aucun accès mémoire	✓	✓	*R*5. retire l'élément de la liste
	– accès statiquement bornés	×	✓	*R*7. remplace par des entrées de type `"m"`
	Pas de modificateur `'&'` superflu	×	✓	*R*8. retire le modificateur `&`

[0] Le compilateur avertira l'utilisateur si l'objet n'est pas initialisé dans le C.
[1] À l'exception du contournement de *clobber* par `push`/`pop`, il s'agit de registre machine, d'appel système, etc...
[2] À condition que le symbole corresponde à une variable globale déclarée dans le C.
[3] Garantie mais très incomplète.

embarqué tel quel. Le bloc est ensuite désassemblé à l'aide de BINSEC [2] et les opérandes sont identifiés à l'aide du DWARF [3].

Cependant, pour garantir les propriétés ***EN*** sur l'ensemble des instanciations possibles du bloc, il est nécessaire de s'assurer que chacune des entrées et sorties ait été correctement identifiée. En effet, une erreur d'identification peut se produire au niveau du binaire si un registre, écrit textuellement dans le code ("en dur", comme `ebx` dans la Fig. 1), est unifié à une entrée ou une sortie valide ou si le compilateur superpose une entrée et une sortie. Ce problème est corrigé en comparant le code obtenu à partir d'instances différentes du bloc : si les objets ne sont pas identifiés de façon identique dans les différentes instances, le bloc viole alors la propriété de ***N***on-ambigüité. Afin d'obtenir des instances où l'allocation de registres est différente, nous nous reposons sur le fait qu'en compilant sans optimisation (option `-O0`), l'ordre des registres sélectionnés est directement relié à celui des éléments de l'interface. Il suffit ainsi d'ajouter un élément ou de faire une rotation de ceux présents – sans connaissance de l'architecture – pour différencier ces instances.

Analyse. Le désassemblage précédent permet de retrouver le graphe de flot de contrôle du bloc, que nous analysons par une forme simple d'interprétation abstraite [1] pour calculer l'ensemble

des opérandes et emplacements lus et écrits à chaque instruction, vérifiant en chaque point que les lectures sont valides (entrée, *clobber* initialisé ou sortie initialisée), que les écritures sont autorisées (*clobber* ou sortie) et qu'aucune entrée n'est lue après qu'une sortie *pouvant* partager le même emplacement – d'après le diagramme de juxtaposition (Fig. 2) – n'ait été mise à jour.

Rustine. Si un problème est détecté par l'analyse, l'outil tentera de proposer une modification de l'interface qui corrigera le problème avant de relancer la procédure sur le bloc. La Table 2 liste les correctifs apportés par RUSTINA ($R0-8$). De manière générale, les rustines sont indépendantes ; il est toutefois possible qu'un correctif un peu lâche de ***BUEN*** entre en conflit avec ***O*** – la correction restant privilégiée dans ce cas. Pour l'heure, seule la rustine $R3$ pourrait être optimisée à l'aide de $R7$.

Diagnostic. Il se peut que le problème détecté ne possède pas de correctif unique. Les symptômes sont alors remontés à l'utilisateur, en utilisant les catégories définies dans la Table 2. L'expertise humaine est en effet nécessaire pour déterminer, en fonction du contexte, l'*intention* du code et appliquer la correction adéquate. La lecture d'un registre *clobber* avant que celui-ci ne soit initialisé est un exemple de problème détecté mais non corrigé – il pourrait s'agir d'un défaut algorithmique comme d'un oubli d'initialisation, auquel cas la valeur resterait toujours inconnue.

5 Expérimentation

Nous avons lancé RUSTINA sur les sources de **204** paquets de la distribution Debian 8.11 contenant de l'assembleur embarqué – incluant des projets comme ffmpeg, ALSA, GMP et libyuv. La Table 3 résume les différents problèmes détectés, selon les catégories listées dans la Table 2 et le nombre de correctifs automatiquement générés proposées. Notre outil a permis, *en 3 minutes*, de détecter 1587 problèmes sur **1289** blocs (41% des blocs) et de générer automatiquement des correctifs pour 1477 d'entre eux, c'est-à-dire 93% des cas. Il nous a également guidés vers des problèmes plus subtils qui peuvent, quant à eux, être corrigés à l'aide de l'expertise humaine comme celui de l'exemple en Fig. 1.

TABLE 3 – Étude empirique de RUSTINA sur de l'assembleur x86 (Debian 8.11)

(a) Statistiques générales

Debian	8.11
# projet	204
# blocs	3091
durée totale	**3min**

(b) Cas de non-conformité

		# détectés	# corrigés
U	Sortie non initialisée	0	0
E	Lecture d'un registre (non SIMD) libre	9	N/A
	`push`/`pop` du registre `ebx`	25	0
	Lecture d'un registre SIMD libre	75	N/A
	Écriture d'un registre SIMD non déclaré	63	63
	Écriture du registre de conditions	1199	1199
	Accès mémoire non déclaré	132	132
N	Lecture d'une entrée altérée	1	0
O	Sortie écriture & lecture non lue	2	2
	Clobber `"memory"` superflu	80	80
	Modificateur `&` superflu	1	1

Nous avons ainsi préparé des correctifs pour 3 projets, ffmpeg, udpcast et ALSA, en prenant en compte les rustines proposées par RUSTINA mais aussi, en tenant compte de leur spécificité, en corrigeant d'autres problèmes, qui sont actuellement hors de portée de l'état d'automatisation de notre outil. Nous avons pour objectif prochain de proposer l'intégration de nos correctifs dans les

projets identifiés, une fois le tri fait – l'automatisation de la génération de correctifs s'accompagne d'une phase, bien manuelle, de soumission de ces derniers.

Nous avons également testé notre approche sur 3 projets ARM : ffmpeg, GMP et libyuv. Les blocs sont cependant beaucoup plus petits – généralement une instruction pour GMP – et avec une sémantique plus simple (architecture RISC). Nous n'avons ainsi pas détecté de problème de conformité – c'est pourquoi les détails de ces expérimentations ne figurent pas ici.

Discussion. Malgré les problèmes de conformité, les codes actuels « fonctionnent » en général comme attendu, notamment grâce aux limitations techniques du processus de compilation. Il est cependant possible de montrer que le compilateur est à même de générer des solutions problématiques aux codes insuffisamment contraints. Cela se produit lorsque les blocs erronés sont plongés dans un contexte différent (réutilisation de code), plus favorable au déclenchement d'optimisations agressives du compilateur : *inlining*, vectorisation, etc... Les menaces, aujourd'hui bénignes, sont donc à prendre au sérieux alors que les compilateurs incorporent des optimisations toujours plus poussées.

6 Conclusion

RUSTINA est un analyseur qui permet aux utilisateurs d'assembleur embarqué de vérifier que leurs spécifications et implémentations coïncident. C'est une analyse peu coûteuse et qui permet de s'assurer que les interfaces d'assembleur embarqué décrivent un code qui interagira de manière correcte avec son environnement C, et notamment qui comporte toutes les informations nécessaires au compilateur pour insérer le bloc dans le code émis. RUSTINA a déjà permis de trouver un certain nombre de cas problématiques d'usage d'assembleur en ligne dans la distribution Linux Debian 8.11, et de proposer automatiquement des correctifs pour ces derniers.

Références

[1] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL. ACM, 1977.

[2] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In SANER. IEEE Computer Society, 2016.

[3] DWARF Debugging Information Format Committee. DWARF Debugging Information Format 5, 2017.

[4] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried. Some assembly required - program analysis of embedded system code. In SCAM. IEEE Computer Society, 2008.

[5] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c : A software analysis perspective. Formal Asp. Comput., 27(3) :573–609, 2015.

[6] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet. De l'assembleur sur la ligne ? Appelez TInA ! In JFLA, 2019.

[7] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet. Get rid of inline assembly through verification-oriented lifting. In ASE. IEEE Computer Society, 2019.

[8] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck. An analysis of x86-64 inline assembly in c programs. In VEE. ACM, 2018.