Analyser du code mixte: C et assembleur embarqué

Frédéric Recoules*, Sébastien Bardin*, Richard Bonichon*, Laurent Mounier† et Marie-Laure Potet†
*CEA LIST, Laboratoire de Sûreté et Sécurité du Logiciel, Paris-Saclay, France
prenom.nom@cea.fr
†Univ. Grenoble Alpes. VERIMAG, Grenoble, France
prenom.nom@univ-grenoble-alpes.fr

***Résumé*—Les méthodes formelles appliquées au logiciel ont fait des progrès importants lors des deux dernières décennies. Leur application à l'embarqué est ainsi un succès indéniable. Parmi les prochains défis réside la question de leur application à des domaines moins contraints. Par exemple, 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. Ceci empêche complètement l'utilisation des méthodes développés pour le C pur. Nous proposons ici TInA, une méthode générale, automatique, sûre et orientée vérification pour pouvoir porter le code assembleur en ligne vers un code C sémantiquement équivalent, et profiter en retour des analyses pré-existantes pour ce langage. Nos expérimentations sur du code C d'envergure montrent la faisabilité et l'intérêt de l'approche.**

I. Introduction

Contexte. Les méthodes formelles utilisées dans le développement de logiciel ont fait des progrès conséquents lors des deux dernières décennies [1], [2], [3], [4], [5], avec des succès notables dans des domaines industriels critiques, souvent régulés, comme l'avionique, le transport ferroviaire ou l'énergie. Cependant, l'application de ces méthodes pour le logiciel disons « courant », moins contraint, que ce soit pour la sûreté ou la sécurité, reste un défi scientifique. En particulier, l'extension des méthodes issues de domaines critiques, avec des règles de codages strict et des processus de validation obligatoires, à des domaines généraux, plus variés et laxes dans les méthodes de développement, est une tâche difficile.

Problème. Nous considérons ici le problème de l'analyse de code « mixte », combinant assembleur en ligne et code C/C++. Cette fonctionnalité, présente dans les compilateurs GCC, clang et Visual Studio, 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. Dans certains cas, il s'agit également d'un moyen d'empêcher le compilateur de réaliser des optimisations non désirées. On la retrouve fréquemment dans les domaines comme la cryptographie, le multimédia ou les pilotes matériel.

Cependant, les analyseurs de programme C/C++ gèrent mal cette fonctionnalité. Certains comme Frama-C [3] la gèrent peu, d'autres comme KLEE [2] ne la gèrent pas du tout. Dans ce cas, l'analyse produira des résultats incorrects ou incomplets. Ceci correspond à un réel problème d'applicabilité des techniques avancées d'analyse de code.

Étant donné qu'il est très coûteux de redévelopper des analyseurs dédiés, la façon usuelle de traiter ces morceaux de code assembleur est de proposer un code équivalent, dans le langage hôte. Cette tâche etant effectuée manuellement, cela met d'emblée hors de portée l'analyse de larges bases de code. Il est en effet chronophage de traduire manuellement de gros bouts d'assembleur, et cela est aussi une source d'erreur non négligeable.

Objectifs. Nous souhaitons concevoir et développer une technique automatique, générique, sûre et orientée vérification pour porter le code assembleur en ligne en code C sémantiquement équivalent, afin de réutiliser les analyses de vérification C/C++. Cette méthode se veut :

Largement applicable Elle ne doit pas être liée à une architecture matérielle, un dialecte d'assembleur ou un compilateur particulier, tout en gérant un large sous-ensemble de l'assembleur en ligne qui se trouve dans les applications courantes ;

Correcte Le processus de traduction doit maintenir exactement tous les comportements du code d'origine, et proposer une façon de démontrer ladite correction ;

Compatible avec la vérification formelle La traduction se doit d'être agnostique vis-à-vis des techniques de vérifications classiques, tout en permettant une qualité d'analyse post-traduction suffisamment bonne en pratique (nous utiliserons le terme de *vérifiabilité*).

Travaux connexes. Les travaux menés jusqu'à présent en vérification de code « mixte » ne remplissent pas tous ces objectifs. Ainsi, Vx86 [6] est ciblé x86 (architecture Intel 32-bits) et vérification déductive mais ne fournit aucun moyen de montrer la correction de la traduction. Au premier abord, les techniques de décompilation [7], [8] peuvent sembler bien adaptées. Mais leur but premier est l'aide à la rétro-ingénierie. De fait, cette famille de techniques peut remettre en cause la correction du processus, à tel point que *« les décompilateurs existants produisent fréquemment un code décompilé qui n'est pas complètement fonctionnellement équivalent au programme d'origine »* [9]. Certains travaux récents [10], [9] ciblent ce critère

de correction mais Schwartz et. al [10] n'étudient pas la question de savoir si le code produit peut être vérifié par les outils actuels et ne démontrent pas la correction de leur approche, se reposant sur du test intensif.

Proposition. Nous proposons TInA (*Taming INline Assembly*), une technique automatique, générique, sûre, orientée vérification pour porter du code assembleur en ligne vers un C équivalent. Un élément clé de TInA est qu'en nous concentrant sur l'assembleur en ligne, plutôt que sur le problème général de la décompilation, nous attaquons un problème plus restreint (taille de code, portée bien définie, type d'instructions et de constructions), mieux défini (grâce à l'interface avec le code C), ouvrant ainsi la porte à une technique ciblée plus puissante. En particulier, TInA est fondé sur les principes suivants :

- La réutilisation de plateformes d'analyse de code exécutable [11], [12], [13] qui permettent de porter du code binaire vers un langage intermédiaire, indépendant de l'architecture matérielle [1], éprouvé et concis.
- Un algorithme de vérification d'équivalence de programmes dédié [2], ramené ici à la résolution d'un problème plus réduit, ciblé pour notre processus.
- Des passes de transformations dédiées, pour améliorer la vérifiabilité de notre résultat, par raffinements successifs du langage intermédiaire brut vers des abstractions le rapprochant de C (flot de contrôle haut niveau, types de données, opérations [14]).

Contribution. En résumé, cet article décrit les contributions suivantes :

- Une méthode (Sec. II) pour ramener l'assembleur en ligne à du code C, améliorée par l'usage du contexte, et validée automatiquement afin de pouvoir faire confiance au processus de traduction mais également de ne pas mettre en péril les critères de correction des analyses formelles considérées.
- Le prototypage de notre méthode atteste de son intérêt pratique (Sec. III) : nous portons et validons 74% des blocs assembleur trouvés dans la distribution Linux Debian 8.11 (Table I) et améliorons les résultats d'analyseurs de l'état de l'art (Table II).

II. Aperçu de TInA

Prototypage. Notre méthode exploite les fonctionnalités d'outils existants comme Frama-C [3], pour son *front-end* C, et Binsec [12] pour traduire le code exécutable en représentation intermédiaire DBA, faire nos analyses et utiliser les solveurs SMT pour la validation.

1. Des instructions en lien avec le matériel, le système d'exploitation ou les flottants ne sont cenpendant pas supportées.

2. Les problèmes de vérification de logiciel sont généralement indécidables — c'est le cas de l'équivalence de programmes. Cela n'empêche bien évidemment pas les outils de vérification d'exister et d'être utiles en pratique.

Le code est compilé et le bloc assembleur est identifié à l'aide du *front-end* C et des informations de débogue. La sémantique est alors extraite en représentation intermédiaire sur laquelle nous pouvons appliquer plusieurs passes de simplifications optionnelles avant d'émettre un code C et de l'insérer à l'ancien emplacement du bloc. L'approche est schématisée en Fig. 1.

Garanties obtenues. La confiance apportée par TInA repose sur une validation du portage plutôt que la validation du traducteur, problème en général plus aisé. Celle-ci vérifie effectivement que les transformations successives faites sur la représentation intermédiaire (IR) préservent la sémantique, autrement dit, que les exécutables obtenus par compilation du code original et du code porté sont sémantiquement équivalents. La correction de la méthode en totalité repose non seulement sur la validation mais également sur une base de confiance comportant le compilateur, le lifteur vers l'IR et le solveur SMT, éléments supposés n'introduire aucune erreur.

III. Expérimentations

Le prototypage de notre méthode atteste de son intérêt pratique sur une base de code de grande ampleur. Nous avons extrait environ 3000 fonctions comportant de l'assembleur x86 en ligne dans la distribution Linux Debian Jessie 8.11. Notre prototype en porter et valide **74%** — **100%** des blocs portés sont validés (cf. Table I pour les chiffres précis). Les 26% restant sont partagés en trois catégories. Les blocs dont la traduction est trivial (vide de mnémonique assembleur) sont laissés tel quels. Les blocs contenant des instructions non supportées par notre plateformes d'analyse de binaire — les instructions manipulant les nombres en virgule flottante ou celles en lien avec le système d'exploitation — sont considérés hors sujet. Enfin, un bloc est jugé dangereux (et rejeté) lorsque l'interface du bloc asm est mal spécifiée.

Table I: Expérimentations sur les morceaux asm de Debian

Debian 8.11 Projets	x86 Total		ARM GMP	
Blocs asm	3039		308	
Vide d'instructions	94		28	
Hors-sujet	486		0	
Rejetés	202		2	
Pertinents	2257	74%	278	90%
Portés	2257	100%	278	100%
Validés	2257	100%	278	100%
# instructions (moyenne / max)	8 / 341		5 / 10	

Des expérimentations additionnelles sur l'architecture ARM et différents compilateurs (différentes versions de GCC ou clang) confirment la généricité de la méthode.

Nous avons également mesuré l'impact de notre portage sur des analyseurs de l'état de l'art. Le greffon EVA (*interprétation abstraite* [15]), par exemple, réalise une analyse de valeurs des variables du code source et lève des alarmes

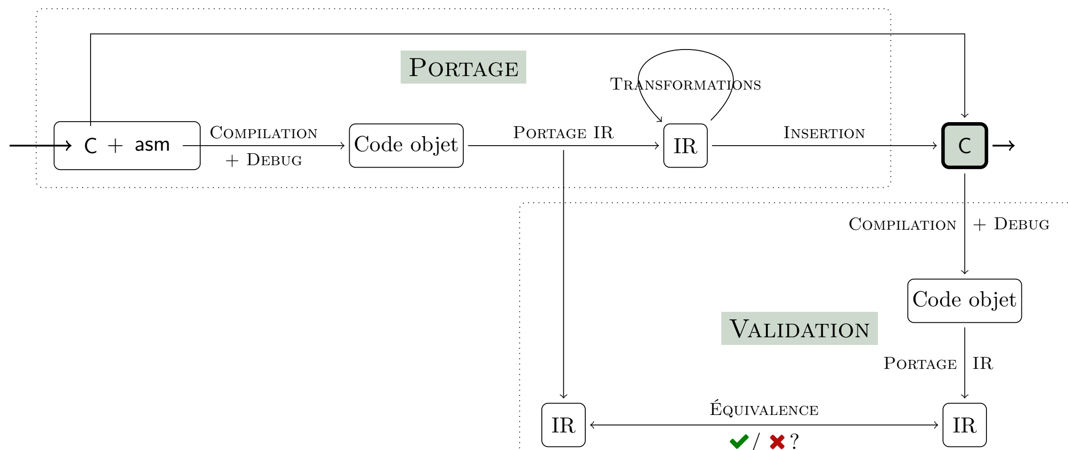

Figure 1: TInA : une vue haut niveau

s'il existe un risque d'erreur à l'exécution (débordement de tableaux, division par zéro, ...). Le code assembleur ne faisant pas partie de la syntaxe du C, il surapproxime sa sémantique – les sorties du bloc peuvent prendre *n'importe quelle valeur*. Notre portage permet de restaurer la précision de l'analyse en dévoilant la sémantique du bloc. Ce regain de précision permet de réduire le nombre de faux positifs parmi les alarmes générées. Cependant le portage n'est pas suffisant pour garantir la *vérifiabilité* : les analyseurs utilisent des abstractions de haut niveau, facilement mises à mal par le code assembleur. La Table II permet d'observer la différence entre une traduction littérale du DBA et sa version raffinée sur un panel de 58 fonctions issues de Debian.

Des expérimentations additionnelles sur KLEE (*exécution symbolique* [16]) et le greffon WP (*vérification déductive* [17]) montrent également un impact positif.

Table II: Impact de la stratégie de portage sur EVA

Portage	Sans	Littéral	TInA
fonctions analysées sans alarmes	N/A	11	20
alarmes émises dans le C original	231	184	177
alarmes émises dans le portage	N/A	316	128

IV. Conclusion

Nous proposons une méthode sûre permettant l'analyse de code C/C++ comportant de l'assembleur en ligne. Cette méthode génère un code C bien structuré permettant la réutilisation de techniques de vérification existantes pour le C, en utilisant des transformations successives sur un langage intermédiaire extrait de l'exécutable. La correction de la méthode est assurée par validation de la traduction. Nos expérimentations sur des codes libres d'envergure écrits en C démontre l'applicabilité de la méthode et son intérêt pratique pour la vérification.

Références

[1] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The static driver verifier research platform," in *Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings.*

[2] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. Proceedings.*

[3] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c : A software analysis perspective," *Formal Asp. Comput.*, 2015.

[4] D. Delmas and J. Souyris, "Astrée : From Research to Industry," in *Static Analysis, 14th International Symposium, SAS 2007. Proceedings.*

[5] P. W. O'Hearn, "From Categorical Logic to Facebook Engineering," in *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015.*

[6] S. Maus, M. Moskal, and W. Schulte, *Vx86 : x86 Assembler Simulated in C Powered by Automated Theorem Proving.* Springer.

[7] C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," *Softw., Pract. Exper.*, 1995.

[8] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to High-Level Language Translation," in *International Conference on Software Maintenance, ICSM 1998.*

[9] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Logino, "Evolving Exact Decompilation," in *BAR 2018, Workshop on Binary Analysis Research.*

[10] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring," in *Proceedings of the 22th USENIX Security Symposium*, 2013.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP : A binary analysis platform," in *Computer Aided Verification - 23rd International Conference, CAV 2011. Proceedings.*

[12] A. Djoudi and S. Bardin, "BINSEC : binary code analysis with low-level regions," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015. Proceedings.*

[13] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[14] A. Djoudi, S. Bardin, and É. Goubault, "Recovering high-level conditions from binary programs," in *FM 2016 : Formal Methods - 21st International Symposium. Proceedings.*

[15] P. Cousot and R. Cousot, "Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[16] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, 1976.

[17] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, 1969.