Vérification automatique de code bas-niveau : C, assembleur et binaire

Frédéric Recoules

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Frédéric RECOULES

Thèse dirigée par **Marie Laure POTET**
et codirigée par **Sebastien BARDIN**, CEA Paris-Saclay
et **Richard BONICHON**, Nomadic Labs

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Vérification automatique de code bas-niveau : C, assembleur et binaire

Automatic Verification of low-level code: C, assembly and binary

Thèse soutenue publiquement le **30 septembre 2021**,
devant le jury composé de :

Monsieur ANTOINE MINÉ
PROFESSEUR DES UNIVERSITÉS, SORBONNE UNIVERSITÉ,
Rapporteur
Madame VALÉRIE VIET TRIEM TONG
PROFESSEUR, CENTRALESUPELEC,
Rapporteure
Madame JULIA LAWALL
DIRECTRICE DE RECHERCHE, INRIA CENTRE DE PARIS,
Examinatrice
Monsieur ROLAND GROZ
PROFESSEUR DES UNIVERSITÉS, GRENOBLE INP,
Président du jury
Madame MARIE-LAURE POTET
PROFESSEUR DES UNIVERSITÉS, GRENOBLE INP,
Directrice de thèse
Monsieur SÉBASTIEN BARDIN
INGÉNIEUR DE RECHERCHE, CEA, LIST,
Invité du jury
Monsieur RICHARD BONICHON
INGÉNIEUR, NOMADIC LABS,
Invité du jury

Abstract

Formal methods for software development have made great strides in the last two decades, to the point that their application in safety-critical embedded software is an undeniable success. Their extension to non-critical software is one of the notable forthcoming challenges. For example, C programmers regularly use GNU style *inline assembly* for low-level optimizations and system primitives. This usually results in rendering state-of-the-art formal analyzers developed for C ineffective. This is particulary problematic since inline assembly is notoriously hard to write correctly: not only the assembly chunk may contain some errors, but there is a risk of a mismatch at the interface between C and assembly, leading to subtle and hard-to-find bugs. We propose to address the problem of verifying C programs containing inline assembly. We thus designed two techniques, named RUSTInA and TInA, based on an original formalization of inline assembly together with novel dedicated algorithms. RUSTInA is the first automated technique for formally checking inline assembly inteface compliance (i.e. no mismatch between code and interface), with the extra ability to propose (proven) patches and code refinements (optimization) in certain cases. TInA is the first automated, generic, verification-friendly and trustworthy lifting technique turning inline assembly into semantically equivalent C code amenable to verification, in order to take advantage of existing C analyzers. Extensive experiments on real-world code (all assembly chunks found on the Debian Jessie packages) raised 986 significant issues in 54 packages, including 156 issues in 7 packages that were succesfully repported to and addressed by the developpers thanks to our automatic patch generation method, and show the feasibility of our principled assembly-to-C lifting and its benefits for state-of-the-art C analyzers.

Résumé

Les méthodes formelles pour le développement logiciel ont fait de grands progrès au cours des deux dernières décennies, au point que leur application dans les logiciels embarqués critiques pour la sûreté est un succès indéniable. Leur application aux logiciels non critiques est cependant l'un des défis majeurs à venir. Par exemple, les programmeurs C utilisent régulièrement l'assembleur embarqué (*inline assembly*) GNU pour réaliser des optimisations de bas niveau ou accéder aux primitives du système. Cela a généralement pour conséquence de rendre inopérants les analyseurs formels de pointe développés pour C. Ceci est doublement problématique puisque l'assembleur embarqué est notoirement difficile à écrire correctement : non seulement le morceau d'assembleur peut contenir des erreurs, mais il y a également un risque d'incompatibilité à l'interface entre C et assembleur, ce qui peut entraîner des *bugs* subtils et difficiles à trouver. Nous proposons d'attaquer le problème de la vérification de programmes C contenant de l'assembleur embarqué. Nous avons donc conçu deux techniques, nommées RUSTInA et TInA, basées sur une formalisation originale de l'assembleur embarqué, ainsi que de nouveaux algorithmes dédiés. RUSTInA est la première technique automatisée pour la vérification formelle de la conformité de l'interface de l'assembleur embarqué (c'est-à-dire l'absence d'incompatibilité entre le code et l'interface), avec la capacité supplémentaire de proposer des correctifs (prouvés) et des raffinements de code (optimisation). TInA est la première technique de traduction automatique, générique, adaptée aux outils de vérification formelle et digne de confiance qui transforme l'assembleur embarqué en code C sémantiquement équivalent, afin de tirer parti des analyseurs C existants. Des expériences intensives sur du code réel (tous les morceaux d'assembleur embarqué trouvés dans les paquets de Debian Jessie) ont permis de remonter 986 problèmes significatifs dans 54 paquets, dont 156 problèmes de 7 paquets qui ont été signalés avec succès et traités par les développeurs grâce à notre génération automatique de correctifs. Ces expériences montrent également la faisabilité de notre traduction « orientée pour la vérification » et ces avantages pour les analyseurs C de l'état de l'art.

Remerciements

Il n'aura suffi que du temps de ce travail pour que *ces* remarques sur *ces* cheveux blancs passent d'anecdotiques à récurrentes. Ce n'est toutefois non sans aucune fierté que je remets entre vos mains le fruit de ce temps investi.

Aussi, et avant de rentrer dans le vif du sujet, je tiens à remercier toutes les personnes qui ont participé, de près ou de loin, à l'accomplissement de ce travail, et sans lesquelles vous ne liriez point ces lignes.

Je remercie tout particulièrement mes encadrants pour le temps et l'énergie qu'ils m'ont consacrés pendant ces longues années. Merci Marie-Laure pour avoir dirigé ma thèse de bons conseils et ce malgré la distance géographique, et pour m'avoir permis de terminer mes travaux à VERIMAG après la fin de mon financement CEA. Merci Richard de m'avoir soutenu, supporté au jour le jour pendant tout mon séjour au CEA et d'avoir poursuivi l'encadrement de la thèse même après avoir changé de poste. Merci Sébastien pour tes innombrables suggestions et relectures. En outre, merci de m'avoir offert l'opportunité de rejoindre l'équipe BINSEC en tant que chercheur permanent.

Je remercie également les coauteurs de mes publications, Laurent Mounier et Matthieu Lemerre, pour leur aide précieuse tant au niveau des idées que pour l'assistance à la rédaction.

Je remercie chacun des membres de mon jury d'avoir pris le temps d'évaluer mon travail – tout particulièrement mes relecteurs y ayant consacré partie de la saison estivale – et d'avoir su animer la soutenance par leurs questions nombreuses et intéressantes.

Je n'oublie pas de remercier toutes les personnes que j'ai côtoyées au sein de mes laboratoires d'accueil, le LSL et VERIMAG, sans oublier tous mes proches, famille, amis et conjointe sans qui la thèse n'aurait été qu'une longue et monotone succession de 0 et de 1.

Enfin je vous remercie vous, lectrices, lecteurs, de donner un sens à mon travail en participant à sa diffusion.

Table des matières

Chapitre 1

Introduction

Nous vivons dans un monde où le *cyber* a pris une place prépondérante. Du divertissement à l'aérospatial, en passant par le transport ou la finance, une majorité de nos activités sont régies par des programmes informatiques. En quelques décennies, l'écosystème de l'informatique est ainsi passé de peu accessible – quelques machines peu abordables et difficiles à manipuler – à omniprésent – l'offre est aujourd'hui large, diversifiée et permet de répondre à une multitude de besoins.

Les révolutions technologiques ont rendu possible cette évolution en augmentant les capacités du matériel et en diminuant les coûts – le nombre de transistors dans une puce a, par exemple, doublé environ tous les deux ans (loi de Moore) pour passer de quelques milliers en 1971 à plusieurs milliards de nos jours.

Le développement logiciel a également participé à cette évolution. L'expérience accumulée au fil des années a permis d'améliorer la formation des métiers de l'informatique. De plus, la base de logiciels, d'outils et de librairies accessibles « clé en main » accélère grandement la création de nouveaux projets. On notera aussi la diversification des langages et des paradigmes de programmation qui permet de mieux s'adapter aux exigences des problèmes ou aux préférences du développeur.

En règle générale, les langages dits de « haut niveau » favorisent l'intelligibilité, la concision et la sûreté (ex. ADA, OCaml, etc.), là où les langages dits de « bas niveau » comme C, sont utilisés pour leur proximité avec le matériel et leur efficacité lors de l'exécution du programme.

Le langage C. Pour des raisons historiques, le langage C est très largement répandu. Il est généralement utilisé dans les couches basses du logiciel et dans les *systèmes embarqués*, où il offre l'un des meilleurs rendements en terme de ressources utilisées [PCR+17].

Le langage C se place comme une abstraction du langage assembleur. Les compilateurs (généralement optimisants tels que GCC, Clang ou VisualStudio) se chargent de convertir un programme C vers la syntaxe assembleur de l'architecture cible. La syntaxe du C libère ainsi le développeur d'une partie de la complexité du programme tel que l'utilisation des mnémoniques assembleur ou encore la gestion des registres. L'organisation structurelle de la mémoire, comprenant les aspects d'allocation dynamique, reste cependant à la charge du développeur. La mauvaise gestion de cette mémoire est l'une des sources d'erreur les plus fréquentes causant des dysfonctionnements dans les logiciels.

Sûreté de fonctionnement et méthodes formelles. Le langage C est utilisé dans l'industrie pour réaliser des tâches sensibles. En effet, un certain nombre de domaines, comme le nucléaire, le ferroviaire, l'avionique ou l'aérospatial sont qualifiés de *critiques* car les répercussions d'un dysfonctionnement peuvent être lourdes de conséquences économiques ou humaines. Aussi, pour s'assurer du bon fonctionnement de ces programmes sensibles, les ***méthodes formelles*** [Att07] sont utilisées conjointement aux pratiques classiques de qualité logicielle (normes de codage strictes, tests unitaires, relectures par des pairs, etc.).

Les méthodes formelles regroupent un ensemble de techniques, telles que l'*interprétation abstraite* [Cou96], la vérification de modèle (*model checking*) [CHVB18] ou encore la vérification déductive, dont le calcul de préconditions (*weakest precondition calculus*) [Dij75], ayant en commun une définition mathématique de la sémantique du programme. Dans le cadre général, les problèmes que cherchent à résoudre les méthodes formelles sont indécidables [Ric53], mais dans la pratique, les outils de vérification se montrent efficaces pour l'audit de code.

Les méthodes formelles ont fait d'énormes progrès au cours des deux dernières décennies et de nombreux outils de vérification automatique ont vu le jour. Pour le langage C, on pourra ainsi citer :

Astrée. [CCF+05] basé sur l'interprétation abstraite et utilisé avec succès par Airbus ;

Frama-C. [KKP+15] plate-forme d'analyse autour de laquelle s'articulent plusieurs greffons dont le greffon EVA (interprétation abstraite) utilisé par EDF ou le greffon WP (vérification déductive) utilisé par Airbus ;

Infer. [CDOY09] développé et utilisé par Facebook pour analyser ses sources. Il a notamment été appliqué sur Mozilla Firefox ;

KLEE. [CDE08] moteur d'exécution symbolique de référence dans le domaine académique. Il a par exemple été utilisé pour analyser des protocoles réseaux du système d'exploitation Contiki OS ;

Polyspace. [Mun12] basé également sur l'interprétation abstraite et utilisé par Nissan, Alenia Aéronautique ou Miracor.

Le compilateur certifié CompCert [Ler09] permet de façon complémentaire de s'assurer qu'un programme C jugé correct au niveau source le restera dans le binaire produit.

Ces outils et une recherche académique très active sur le sujet contribuent à la promotion des méthodes formelles dans le secteur du développement logiciel. Toutefois, ces outils sont bien souvent limités à un sous-ensemble du langage C, comme par exemple, le C ANSI. Or, dans le développement logiciel non critique (moins régulé) le C apparaît plus compliqué et peut se mélanger avec d'autres langages.

Nous allons nous intéresser spécifiquement à l'**assembleur embarqué** (*inline assembly*), une extension des compilateurs qui permet d'intégrer directement des instructions assembleur dans ses sources C.

Assembleur embarqué. L'assembleur embarqué est une déclaration particulière du langage C permettant d'écrire directement des mnémoniques assembleur au milieu de code C. Cette pratique permet d'utiliser directement des primitives du jeu d'instructions du processeur, en particulier lorsque le compilateur n'est pas capable de les générer automatiquement. Les compilateurs ont pour mission d'émettre le code assembleur inchangé ce qui permet, à l'instar d'une fonction écrite en assembleur pur, de contrôler précisément le résultat final (*What You See Is What You Get*). Ce contrôle est alors utilisé pour

— accéder au matériel (*hardware*) ;

— optimiser manuellement un code pour un processeur donné ;

— réaliser des opérations que le compilateur ne doit pas altérer, comme par exemple le nettoyage après une section critique de cryptographie.

Les normes C (C99 et suivantes) spécifient la forme d'un bloc assembleur (mot clé `asm`) mais laissent la liberté au compilateur d'en définir son contenu. En pratique il existe deux syntaxes largement utilisées et supportées par les compilateurs contemporains :

Syntaxe Microsoft : proposée par VisualStudio et supportée par ICC ;

Syntaxe GNU : proposée par GCC et supportée par Clang, ICC ou encore, dans une moindre mesure, CompCert.

Dans les deux cas, la présence de code étranger peut perturber sévèrement le fonctionnement des outils de vérification de code C. Ainsi, un support inexistant, comme c'est le cas pour KLEE, mènera l'analyse à s'arrêter. Un support limité, à la Frama-C, est susceptible de rendre les analyses imprécises et leurs résultats inexploitables. Dans le pire des cas, l'outil pourra faire des suppositions rendant le résultat de ses analyses incorrect.

Les deux syntaxes proposent des approches sensiblement différentes et ne soulèvent pas les mêmes problématiques. Outre la syntaxe à proprement parler, la principale différence est que chez Microsoft, le compilateur supporte nativement un sous-ensemble d'instructions assembleur (limité à l'architecture x86 32 bit) et est capable de les lier automatiquement avec son contexte C. Pour GNU, l'assembleur est passé sous forme de chaînes de caractères, ce qui le rend plus générique mais également plus difficile à écrire correctement. Notamment, le développeur a la charge d'écrire l'interface qui liera l'assembleur au C, le compilateur se reposant aveuglément sur cette interface pour compiler le code.

La vérification de codes mixtes, C et assembleur, a déjà fait l'objet de travaux de recherche à Microsoft, en particulier avec la thèse de Maus [Mau11]. Les résultats ne sont pourtant pas applicables directement aux codes utilisant la syntaxe GNU.

La vérification de code utilisant l'assembleur embarqué de GNU reste donc un problème ouvert.

L'assembleur embarqué est pourtant une pratique assez courante dans le milieu du développement logiciel sous Linux. En 2018, Rigger et al. [RMK+18] publiait une étude annonçant que **28**% des projets GitHub les plus visibles écrits en C contient

de l'assembleur embarqué. Nous estimons, pour notre part, qu'environ **11**% des paquets de la distribution Debian Jessie en contient également. Si l'on s'attend à ce que l'assembleur soit principalement transmis par héritage, le fait est qu'une base de code importante contient encore aujourd'hui de l'assembleur embarqué au sein de ses sources. Il s'agit donc d'autant de projets qui restent hors de portée de la vérification par les méthodes formelles.

Objectif. *Dans cette thèse, notre objectif est de rendre l'utilisation d'assembleur embarqué plus sûre en concevant et développant des techniques automatiques permettant la vérification de codes mixtes C et assembleur embarqué GNU.*

En particulier, ces techniques devront :

— s'adapter aux spécificités de la syntaxe GNU et détecter les problèmes qui y sont liés ;

— être génériques et pouvoir s'appliquer facilement à l'assembleur rencontré, quel que soit son jeu d'instructions ;

— favoriser l'interopérabilité avec les techniques et outils C existants.

Conjointement, elles devront être fiables afin de pouvoir être utilisées dans un contexte de vérification formelle en toute confiance.

L'assembleur nous force à quitter le niveau d'abstraction offert par le langage C. À l'inverse, descendre au niveau de l'analyse de code binaire peut se révéler hasardeux. En effet, les analyses de niveau binaire perdent en précision plus rapidement et ont des difficultés à passer à l'échelle. De plus, l'interface avec le C étant perdue à la compilation, il serait impossible de détecter un problème concernant le lien entre les deux langages.

Il est donc nécessaire de travailler à un niveau intermédiaire, hybride, qui n'a pas encore été étudié.

Difficultés et défis. La conception de ces nouvelles techniques soulève 3 grands défis :

— la sémantique du C mélangé à de l'assembleur embarqué n'est pas définie, que ce soit au niveau opérationnel ou concernant la nature du contrat constituant l'interface ;

— il n'existe pas de définition formelle de ce que doit être la conformité entre le code assembleur et son interface ;

— les techniques qui ont fait leur preuve sur C reposent sur des abstractions fournies par C, absentes au niveau de l'assembleur.

La forme GNU de l'assembleur embarqué n'a été que peu étudiée. Fehnker et al. [FHRS08] ont publié, à notre connaissance, la seule tentative pour inspecter la conformité de l'interface. Leur définition est toutefois incomplète et omet certains problèmes de compilation liés à l'interface pourtant connus des développeurs. En outre, leur technique paraît fragile étant donné qu'elle se base sur des aspects syntaxiques de l'assembleur ARM. Aussi, bien qu'ils affirment que leur technique est

adaptable à d'autres architectures, possiblement plus complexes comme la famille x86, l'effort d'ingénierie nécessaire se révèle bien trop conséquent pour être viable.

Les tentatives précédentes de vérification de code combinant C et assembleur ne remplissent pas tous nos objectifs. Ainsi, les travaux de Maus [Mau11] opèrent sur la syntaxe Microsoft de plus haut niveau que celle de GNU. De plus, nos expérimentations montrent que la traduction qu'ils proposent, si elle donne de bon résultat avec l'outil de vérification déductive VCC [Sch08], a tendance à être de trop bas niveau pour d'autres outils tel que le greffon WP de Frama-C. Il en est de même pour les travaux de Corteggiani et al. [CCF18] qui se concentrent sur le moteur d'exécution symbolique KLEE. Par ailleurs, l'interface de l'assembleur embarqué n'est pas considérée : elle est uniquement concrétisée pour produire de l'assembleur, ce qui rend difficile, si ce n'est impossible, sa vérification a posteriori. Toute approche se basant sur la décompilation du code après compilation souffrirait ainsi du même problème.

Aussi, sans même parler des problèmes posés par l'interface GNU, aucune approche ne se montre suffisamment générique pour aborder à la fois la pluralité des architectures et des techniques de vérification.

Notre approche. Nous proposons un ensemble de techniques automatiques pour aider la vérification de code C contenant de l'assembleur embarqué GNU et qui se veut générique, fiable et compatible avec les outils de vérification de code C existant. Ces techniques réalisent 3 tâches principales :

- extraire une représentation intermédiaire de l'assembleur embarqué dérivée de celle du code binaire ;
- vérifier que le comportement des instructions assembleur coïncide avec le comportement déclaré par l'interface et, le cas contraire, proposer automatiquement un correctif ;
- si le bloc assembleur est bien formé, traduire ce comportement dans la syntaxe du C, en veillant à conserver ou retrouver des constructions de haut niveau facilitant le travail de vérification avec les outils de niveau source.

En s'attaquant spécifiquement à l'assembleur embarqué, il est possible de réutiliser les techniques de désassemblage et d'extraction sémantique proposées par les plateformes d'analyse de binaires (*binary lifter*) tel que BINSEC [DB15], BAP [BJAS11] ou angr [SWS+16], ceci dans un cadre restreint, tant au niveau de la taille (quelques centaines d'instructions au maximum) que par l'absence de saut dynamique qui représente la difficulté principale pour un désassembleur. L'accès aux sources et aux contraintes de l'interface nous permet de contrôler l'affectation des registres et des adresses mémoire donnés à l'assembleur. Un processus itératif permet alors d'identifier précisément et avec certitude la position des opérandes assembleur provenant de l'interface. Le code peut alors être analysé de façon générique, indépendamment de l'instance qui aurait été choisie par un compilateur. Nous réutilisons ici l'effort d'ingénierie et de tests déjà réalisé par les plateformes d'analyse de binaires, afin de gagner le support de multiples architectures sans avoir à réimplémenter nos propres analyseurs syntaxiques de zéro (*parser*).

L'accès à notre représentation intermédiaire ouvre ensuite la voix aux différentes techniques de vérification basées sur des analyses de flots de données (*dataflow analysis*). Il est ainsi possible d'inférer l'interface minimale nécessaire à la bonne coopération de l'assembleur et du C et de vérifier que l'interface déclarée dans les sources inclut correctement tous ces éléments.

Finalement, la représentation intermédiaire peut être transformée à l'aide de passes adaptées à la nature bas niveau de l'assembleur et d'autres qui s'inspirent des techniques utilisées par les compilateurs pour optimiser du code. Les détails de l'architecture sont éliminés et le code réagencé afin que le code C obtenu par traduction soit plus proche d'un code C naturel, facilitant le travail des outils de vérification.

Contributions. En résumé, ce document fait état des contributions suivantes :

— une sémantique opérationnelle de l'assembleur embarqué dans le cadre du C ; la conception d'une méthode combinant éléments existants (assembleur standard, plateforme d'analyse de binaire, etc.) et modules originaux afin d'extraire automatiquement une représentation intermédiaire riche et générique des instructions assembleur embarquées dans le contexte C ; l'implémentation de cette méthode dans le prototype TINA (environ 3kLOC OCaml) et son application sur l'ensemble des blocs assembleurs x86 rencontrés dans le code de la distribution Debian *Jessie* (environ 3000 blocs) avec un taux de réussite de **85**% ainsi que des expérimentations sur ARM démontrant la généricité de l'approche ;

— une formalisation inédite des problèmes de conformité d'interface généralisant les *bugs* de compilation connus ainsi qu'une méthode automatique de vérification de cette conformité, permettant la détection de blocs assembleurs à risque et la génération automatique de correctifs ; l'extension RUSTINA du prototype TINA (environ 1kLOC) et son application sur le corpus de blocs x86 mettant en évidence la présence de problèmes de conformité à l'interface dans plus d'une centaine de projets open-source (2183 problèmes dont 986 sont jugés sérieux dans un total de 202 projets) et la proposition de correctifs pour 8 projets sélectionnés, dont ALSA, libtomcrypt ou x264 – 7 ont déjà appliqué les correctifs corrigeant 156 problèmes sérieux ;

— une nouvelle méthode de traduction automatique de code assembleur vers le C axée sur la qualité du code généré à des fins de vérification ; une passe de validation dédiée a posteriori permettant de s'assurer de la correction de la traduction afin de garantir son adéquation dans un contexte de vérification formelle ; l'implémentation de cette méthode dans le prototype TINA (environ 2kLOC) et son application sur le corpus de blocs x86 et ARM ; des expérimentations avec les outils de vérification KLEE, Frama-C EVA et Frama-C WP montrant une amélioration significative des résultats d'analyses, respectivement sur les chemins explorés, la précision ou les propriétés démontrées, avec le code traduit par TINA par rapport à une traduction standard.

En conclusion, cette thèse identifie les problèmes et difficultés liés à l'utilisation de code assembleur embarqué dans du code C. Nous sommes les premiers à résoudre les problèmes de conformité à l'interface – avec un impact concret sur la base de code en libre accès – et à aborder le thème de la *vérifiabilité* de code. L'ensemble des méthodes automatiques proposées est implémenté dans le prototype TInA développé en OCaml (environ 5kLOC) au-dessus des plateformes d'analyse de binaire BINSEC et de code C Frama-C.

Publications. Les travaux présentés dans cette thèse ont été publiés à :

ASE19 [RBB+19] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier et Marie-Laure Potet. *Get rid of inline assembly through verification-oriented lifting.* *The 35th IEEE/ACM International Conference on Automated Software Engineering* (rang *core* A) ;

ICSE21 [RBB+21] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Matthieu Lemerre, Laurent Mounier et Marie-Laure Potet. *Interface compliance of inline assembly : Automatically check, patch and refine.* *The 43rd International Conference on Software Engineering* (rang *core* A+).

Une version journal étendue est en préparation pour une soumission à *IEEE Transactions on Software Engineering*.

Exposés et présentations. Ces travaux ont été présentés aux *Journées Francophones des Langages Applicatifs* (JFLA19 et JFLA20), au Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI19) et ont été retenus au *KLEE workshop* 2021. Ils ont été présentés sur invitation au séminaire du *laboratoire d'informatique de Sorbonne Université* (LIP6), de l'*Université de Paris 7*[1] ainsi qu'au *Cyber Festival 2019* organisé par l'ANSSI.

Organisation du document. La thèse est organisée comme suit :

Chapitre 1. La présente introduction générale ;

Chapitre 2. Une introduction didactique à l'assembleur embarqué, illustrant les différents problèmes traités par les contributions cette thèse ;

Chapitre 3. Une introduction du cadre de la thèse et des connaissances générales nécessaires. La section 3.1.1 présente trois des principales techniques de vérification formelle, à savoir l'interprétation abstraite, l'exécution symbolique et la vérification déductive. Les difficultés liées à l'application de ces techniques sur un programme de bas niveau seront discutées dans la section 3.1.2. Finalement, la section 3.2 contient les informations primordiales pour comprendre le fonctionnement de l'assembleur embarqué ainsi qu'une étude empirique de son utilisation dans le développement logiciel ;

1. Présentation donnée par Richard Bonichon.

Chapitre 4. Un chapitre introduisant les notions nécessaires aux chapitres 5 et 6. La section 4.2 définit une représentation intermédiaire du langage assembleur sur la base classique de celle du langage binaire et formalise sa sémantique opérationnelle. Sur cette base, la section 4.3 propose un formalisme pour l'assembleur embarqué, ainsi que son interaction avec le C en définissant la sémantique opérationnelle lorsque le bloc est correctement compilé (i.e. en l'absence de problème défini dans le chapitre 5). Enfin, la section 4.4 propose et implémente une approche pour extraire la représentation intermédiaire de l'assembleur embarqué à partir d'une combinaison d'outils existants et de composants simples à réaliser ;

Chapitre 5. Un chapitre définissant la notion de conformité à l'interface, une méthode pour la vérifier et, en cas d'échec, la corriger automatiquement. La section 5.1 définit formellement les deux propriétés de la conformité à l'interface : le respect du cadre (de l'anglais *framing condition*) et la condition d'unicité. La section 5.2 décrit la technique de vérification automatique de ces propriétés. La section 5.3 décrit comment corriger l'interface pour garantir ces deux propriétés. Enfin, la section 5.5 évalue le prototype implémentant ces techniques sur le corpus d'assembleur présenté en section 3.2 ;

Chapitre 6. Un chapitre décrivant une approche pour permettre la vérification de code mixte (C et assembleur embarqué) en traduisant l'assembleur vers du C, le plus « naturel » possible. La section 6.2 introduit les bases de la technique de traduction. Cette approche ne donnant pas de bons résultats, la section 6.3 introduit un ensemble de simplifications permettant d'obtenir un C plus facile à traiter par les analyseurs automatiques. La section 6.4 discute de la validité de cette traduction dans un contexte de vérification formelle et propose une technique permettant de valider rapidement et efficacement chaque traduction. Enfin, la section 6.5 évalue le prototype TInA sur le corpus d'assembleur (section 3.2) ;

Chapitre 7. La conclusion du document, proposant un résumé des résultats obtenus ainsi qu'une ouverture sur de futurs travaux impliquant TInA.

Chapitre 2

Exemples et motivations

Sommaire

Ce chapitre sert d'introduction à la problématique de l'assembleur embarqué dans du code C. Nous allons voir à travers une série d'exemples les différents problèmes qui peuvent survenir lors de la compilation ainsi que les problèmes qui se posent lors des tentatives infructueuses de vérification du code.

En premier lieu, nous introduisons dans la section 2.1 le processus de compilation, les optimisation à base d'assembleur et les avantages à utiliser la syntaxe embarquée.

Cette syntaxe ne présente cependant pas que des avantages et il est facile de se tromper et d'obtenir des comportements incohérents et non désirés. Dans la section 2.2, nous découvrirons sur deux morceaux d'assembleur embarqué des exemples de défauts d'interface et les problèmes qui leur sont associés.

Enfin, nous illustrerons dans la section 2.3 le comportement de trois outils de vérification face à un bloc assembleur. Les résultats étant insatisfaisants, nous verrons comment pallier les difficultés rencontrées, moyennant des efforts manuels supplémentaires.

Nous terminerons en section 2.4 par une présentation succincte des points clés de notre approche.

2.1 Rencontre avec l'assembleur embarqué

Les processeurs de nos matériels informatiques ne sont que des interpréteurs d'un jeu d'instructions dont les variantes les plus répandues aujourd'hui sont les familles x86 (ordinateur) et ARM (ordinateur, *smartphone*, objet connecté). Une procédure algorithmique se compose d'une série d'instructions encodée sous la forme d'une chaîne de bits dans la mémoire. Manipuler manuellement cette représentation binaire dans l'optique de programmer un processeur est une tâche particulièrement laborieuse. Les langages de programmation permettent de s'abstraire de cette tâche en délégant une partie de la complexité à des outils automatisés. Les algorithmes écrits dans ces langages peuvent être :

Interprétés. le processeur exécute un programme qui va lire le code source ou une de ses représentations dérivées et en exécuter l'algorithme ;

Compilés. le programme est encodé dans le jeu d'instructions du processeur qui exécutera directement le code produit.

Orthogonalement, l'algorithme peut être transformé à différents degrés pour en réduire sa complexité et ainsi réduire sa taille et/ou sa durée d'exécution.

Le langage C. Le langage C est généralement utilisé dans un contexte où les performances sont prioritaires. C'est un langage de bas niveau ayant pour vocation d'être transformé en assembleur (*compilation*) puis en code exécutable natif (*assemblage*). Le cycle de compilation standard d'un programme C est illustré dans la figure 2.1.

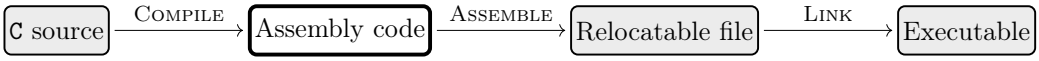


Figure 2.1 – Du langage C à l'exécutable

Les compilateurs de C sont réputés pour favoriser les optimisations. À titre d'exemple, Pereira et al. [PCR+17] ont mesuré des différences significatives dans l'utilisation des ressources en temps et en espace pour différents langages. Un programme écrit en C était ainsi en moyenne 2 à 3 fois plus rapide que le même programme écrit en OCaml. Un facteur 70 a même été observé entre C et Python.

Les compilateurs restent toutefois limités et font de leur mieux (*best effort*) pour que le programme s'exécute le plus efficacement. Il existe donc des situations dans lesquelles un expert est capable de proposer une version du code directement écrite en assembleur plus efficace que celles provenant de la compilation d'un code C.

Nous allons illustrer cette situation avec un exemple inspiré d'une fonction de l'outil FFMPEG : `mid_pred` qui retourne la valeur centrale entre 3 entiers. Cette fonction est utilisée dans l'encodage de vidéo selon le standard H.263 où elle est appelée à répétition sur des flux de données. L'efficacité de cette fonction apparaît donc primordiale pour les performances générales de l'application et nous pensons que cela a justifié qu'elle soit optimisée avec de l'assembleur embarqué x86.

Naïvement, cette fonction pourrait s'écrire très simplement en C comme suit :

```
int mid_pred (int a, int b, int c)
{
  if ((a <= b) && (b <= c)) return b;
  if ((b <= a) && (a <= c)) return a;
  return c;
}
```

Cette première version présente l'inconvénient d'utiliser des sauts conditionnels. Les processeurs de la famille x86 sont conçus pour travailler de façon optimale en séquentiel (*pipeline*) et l'imprédictibilité des sauts conditionnels détériore la vitesse d'exécution.

Optimisation avec l'assembleur. L'architecture x86 possède l'instruction `cmov` qui réalise une affectation conditionnelle sans avoir recours à des sauts conditionnels. Cette instruction peut être utilisée pour réaliser plus efficacement la fonction `mid_pred`. Le compilateur n'utilisant pas (encore) cette instruction, il est nécessaire d'écrire la fonction en assembleur :

```
mid_pred:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        pushl   %ebx
        movl    8(%ebp), %edx
        movl    16(%ebp), %ecx
        movl    %eax, %ebx
        cmpl    %eax, %edx
        cmovg   %edx, %ebx
        cmovg   %eax, %edx
        cmpl    %ecx, %edx
        cmovl   %ecx, %edx
        cmpl    %edx, %ebx
        cmovg   %edx, %ebx
        movl    %ebx, %eax
        popl    %ebx
        popl    %ebp
        ret
```

Cette nouvelle version est plus compacte avec seulement 18 instructions (au lieu de 24, pour la version C compilée avec `-m32 -O3`). Elle est de plus entièrement séquentielle grâce à l'utilisation de `cmov`. Pourtant, l'utilisation de fonctions écrites en assembleur présente deux inconvénients majeurs :

1. le compilateur ignore ce que fait la fonction et ne sera pas capable d'optimiser le code l'appelant ;
2. le compilateur ne peut pas spécialiser la fonction (*inlining*) et devra se plier à la convention d'appel. Si l'on ne considère que le prologue et l'épilogue de la fonction, déjà 7 instructions (soit plus de $\frac{1}{3}$ du nombre d'instruction total de la fonction) ne participent pas au résultat.

La syntaxe de l'assembleur embarqué permet de pallier ce second point en permettant au compilateur de spécialiser le morceau d'assembleur dans son contexte.

```
74  static inline av_const int mid_pred(int a, int b, int c)
75  {
76      int i=b;
77      __asm__ (
78          "cmp    %2, %1 \n\t"
79          "cmovg  %1, %0 \n\t"
80          "cmovg  %2, %1 \n\t"
81          "cmp    %3, %1 \n\t"
82          "cmovl  %3, %1 \n\t"
83          "cmp    %1, %0 \n\t"
84          "cmovg  %1, %0 \n\t"
85          :"+&r"(i), "+&r"(a)
86          :"r"(b), "r"(c)
87      );
88      return i;
89  }
```

FIGURE 2.2 – `libavcodec/x86/mathops.h`@659df32

Intégrer l'assembleur dans le C avec la syntaxe GNU. La fonction `mid_pred` telle que présente dans les sources de FFMPEG est donnée en figure 2.2. Les détails de la syntaxe du bloc assembleur embarqué seront détaillés en section 3.2.2. Pour l'instant, nous pouvons remarquer que le morceau d'assembleur embarqué se compose de plusieurs sections séparées par le signe `':'`. La première section (lignes 78 à 84) contient les instructions assembleur sous forme de chaînes de caractères. La seconde section (ligne 85) contient la liste des sorties du bloc alors que la troisième section (ligne 86) contient la liste des entrées. Ces deux dernières jouent le rôle d'une convention d'appel personnalisée qui permettra au compilateur de faire des choix en fonction de ce qu'il juge le plus intéressant. Ainsi, la lettre `'r'` précédant les variables `i`, `a`, `b` et `c` signifie que le compilateur doit placer ces variables dans des registres généraux sans pour autant imposer le choix exact de ces registres. Lorsque le compilateur aura fait son choix, il remplacera les jetons (*token*) `%0`, `%1`, `%2` et `%3` apparaissant dans les instructions par les registres qu'il aura choisis pour, respectivement, `i`, `a`, `b` et `c`. Il se contentera ensuite d'imprimer, telles quelles, les instructions assembleur dans le fichier assembleur qu'il produit. Le compilateur est également en charge de générer les instructions nécessaires pour que les entrées du bloc soient correctement initialisées et que les résultats soient mis à jour. Si les optimisations sont activées, le compilateur essaiera de minimiser cette « glue » en choisissant les registres dans lesquels ces variables sont déjà présentes.

L'assembleur embarqué permet de faire tout ce qu'un appel à une fonction écrite en assembleur peut faire avec un certain nombre d'avantages :

— il est possible de se passer de la convention d'appel, ce qui est très intéressant pour les petits blocs ;

— le code est plus facile à écrire pour le développeur qui n'a pas besoin de se soucier de la convention d'appel ;

— le compilateur gagne plus de flexibilité dans le choix de l'allocation de registres.

Toutefois, le compilateur ignore ce que le bloc assembleur fait, ce qui peut avoir pour effet de limiter l'optimisation du code qui l'entoure. Plus problématique, le compilateur n'a aucun moyen de savoir si l'interface entre le C et l'assembleur est cohérente avec ce que fait réellement ce dernier. Le compilateur est ainsi forcé de faire confiance aux déclarations du développeur. Nous allons voir dans la section suivante que la syntaxe GNU est réputée pour être difficile [Sta18] et que les erreurs et oublis peuvent provoquer de sérieux bugs.

2.2 Problèmes de compilation

L'assembleur embarqué est plus flexible et, en donnant plus de liberté au compilateur, potentiellement plus efficace que l'assembleur classique. Cette fonctionnalité est toutefois à réserver aux experts. Il est en effet très facile de se tromper en écrivant l'interface entre le langage C et l'assembleur. Le compilateur se reposant entièrement sur les déclarations de l'utilisateur, il n'offre ainsi aucun filet de sécurité. La complexité de cette nouvelle syntaxe s'ajoute donc à la complexité déjà élevée d'écrire de l'assembleur. Plus préoccupant, un bloc assembleur incorrectement déclaré peut, à l'instar des comportements indéfinis du C, fonctionner parfaitement pendant de longues années avant de provoquer un bug. Ces changements peuvent survenir sans qu'aucune modification ne soit faite sur le bloc lui-même. Aussi, mettre à jour le compilateur ou changer une option de compilation peut devenir un élément déclencheur de bugs et il est alors très difficile d'en retracer l'origine.

Nous allons illustrer ces problèmes avec deux exemples issus de codes réels :

1. la librairie libatomic_ops qui fournit des accès aux primitives matérielles d'opérations atomiques. Elle a été utilisée dans le développement de programmes concurrents (*multithreads*). On lui préférera les types atomiques standards à partir de C11 ;

2. la librairie standard libc qui fournit la majorité des fonctionnalités nécessaires aux programmes écrits en C. En particulier, les primitives sur les chaînes de caractères sont optimisées à l'aide d'assembleur embarqué jusqu'à la version 2.19 (fin de vie en juin 2020).

Ces deux exemples historiques montrent des problèmes entre le code assembleur et son interface. Ces problèmes peuvent mener à des bugs et ont été corrigés dans les versions actuelles. Nous verrons en section 5.5 que ce ne sont malheureusement pas des exemples isolés, et qu'un ensemble conséquent (environ 10%) des blocs rencontrés dans le code de la distribution Debian souffrent d'erreurs similaires.

2.2.1 Oubli d'un effet de bord dans libatomic_ops

L'extrait de code de la figure 2.3 est tiré du code source de libatomic_ops datant de fin 2005 (révision 178cc98). L'interface qui lie l'assembleur au C omet de déclarer un effet de bord sur un des registres d'entrée. Bénin à première vue, nous allons nous pencher sur un contexte d'utilisation où ce problème peut provoquer un blocage complet (*deadlock*). Le bloc a été corrigé dans la révision 03e48c1 en 2010, soit 4 années après avoir été introduit.

```
122  /* Returns nonzero if the comparison succeeded. */
123  AO_INLINE int
124  AO_compare_and_swap_full(volatile AO_t *addr,
125                           AO_t old, AO_t new_val)
126  {
127    char result;
128    __asm__ __volatile__("lock; cmpxchgl %3, %0; setz %1"
129                         : "=m"(*addr), "=q"(result)
130                       : "m"(*addr), "r" (new_val), "a"(old) : "memory");
131    return (int) result;
132  }
```

FIGURE 2.3 – atomic_ops/sysdeps/gcc/x86.h@178cc98

En quoi consiste le code. La fonction `AO_compare_and_swap_full` implémente la primitive atomique standard d'« échange si inchangé » (*Compare and Swap*). Cette opération permet, dans un contexte concurrent, de mettre à jour une valeur uniquement si on a la main. Fonctionnellement, `AO_compare_and_swap_full` équivaut à :

```
if (*addr == old) *addr = new_val;
```

Pour ce faire, le bloc assembleur utilise le préfixe `lock` couplé à l'instruction `cmpxchg` dont le pseudocode est (où $\overset{c}{\longleftarrow}$ est l'affectation conditionnelle en fonction de la valeur de c) :

```
lock
cmpxchgl new_val, addr
```

```
        z  ⟵  %eax = *(addr)
    %eax   ⟵  *(addr)
*(addr)   z⟵  new_val
```

Aussi revenons-en à notre bloc assembleur. Les entrées et sorties du bloc sont respectivement déclarées en lignes 130 et 129. Les chaînes de caractères placées devant les expressions C (entre parenthèses) définissent les contraintes d'allocation : `"m"` pour mémoire, `"r"` pour registre général, `"a"` pour le registre `%eax` et `"q"` pour un des 4 registres `%eax`, `%ebx`, `%ecx` et `%edx`.

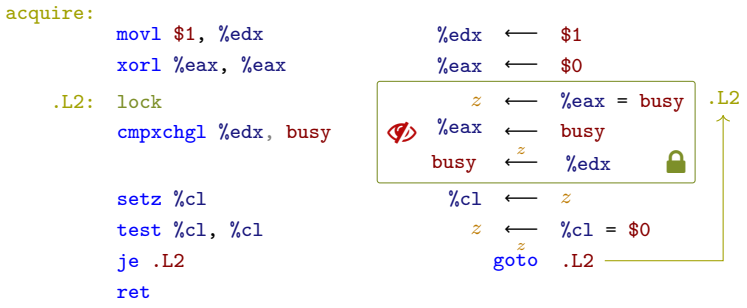
La variable `old` est donc correctement placée dans le registre `%eax` pour obtenir le comportement désiré.

L'omission. La variable `old` n'est toutefois déclarée qu'en « lecture-seule » et seules les variables `addr` et `result` sont déclarées comme étant des sorties du bloc (signe `'='`). Or il apparaît ici que `%eax` peut être modifié par l'instruction si sa valeur est différente de celle présente en mémoire. Ce registre étant utilisé de manière implicite par l'instruction, il n'est pas si évident, en lisant la mnémonique, de savoir ce qui peut lui arriver. Si le programme réutilise la valeur de la variable `old` par la suite et que le compilateur choisit de le faire depuis le registre `%eax`, il risque d'utiliser une mauvaise valeur.

Exemple jouet. La fonction `AO_compare_and_swap_full` peut servir à gérer l'accès à une ressource entre différents *threads* en remplissant le rôle de *mutex* :

```
#include <atomic_ops.h>
volatile int busy = 0;
void acquire () { while (!AO_compare_and_swap_full(&busy, 0, 1)); }
```

Regardons comment GCC compile la fonction `acquire` avec optimisation. Le code assembleur est représenté à gauche et le pseudocode associé à droite :



Le problème. Les arguments de la fonction étant (censés être) constants (`0` et `1`), le compilateur peut décider de sortir l'initialisation des registres `%edx` (`new_val`) et `%eax` (`old`) de la boucle. Observons maintenant les conséquences de ce choix pour `%eax` – que l'on sait être écrasé par l'instruction `cmpxchg`. Normalement, tant que le *mutex* est verrouillé par un *thread* extérieur (`busy == 1`), la comparaison à `0` doit échouer. Ce sera effectivement le cas au premier tour de boucle. Toutefois, la comparaison entre `%eax` et `busy` met également à jour la valeur d'`%eax`. Aussi, au tour de boucle suivant, si la valeur de `busy` n'a pas changé entre temps, la comparaison se résoudra en une égalité et le code pourra continuer sans être bloqué. *Ce défaut de synchronisation entre threads risque de mener à des bugs ou à un blocage (deadlock).*

Type de problème. Le compilateur ignore le contenu des instructions assembleur et se repose donc entièrement sur les déclarations du développeur. Pourtant, le compilateur réalise des optimisations sur le code entourant le bloc assembleur, incluant

les instructions liées à l'interface. Dans le cas général, le compilateur fait des hypothèses conservatrices lorsqu'il rencontre un code inconnu. L'assembleur embarqué au contraire est spécialement pensé pour l'optimisation. L'interface est ainsi présumée complète et seuls les effets déclarés sont pris en compte par les différentes passes d'optimisation. Lorsqu'une information est manquante, le compilateur peut être amené à prendre de « mauvaises » décisions qui perturbent le comportement attendu du programme. Nous formalisons cette catégorie de problème, nommée **non-respect du cadre** (*framing condition*), en section 5.1.2.

Pourquoi le code semble fonctionner. Ce type de problème est d'autant plus dangereux et difficile à corriger qu'il aura tendance à fonctionner correctement sans déclencher de bug. En effet, un bug n'aura lieu que si le compilateur « exploite » la liberté offerte par une information manquante dans l'interface et prend une décision « malvenue ». On parle alors de bug *latent*.

Les compilateurs ayant fait des progrès significatifs, le risque de déclenchement d'un bug latent a cependant augmenté, rendant les codes hérités (*legacy*) d'autant plus dangereux de nos jours.

2.2.2 Mauvaise allocation de registres dans la libc

La libc contient un grand nombre de primitives manipulant des chaînes de caractères. Il y a encore quelques années, ces primitives étaient optimisées à l'aide d'assembleur embarqué pour profiter d'instructions matérielles spécialisées. Le code de la fonction `strcspn` de la figure 2.4 est extrait des sources de glibc *2.19* présentes dans la distribution Debian *Jessie* (2015-2018). Sur la même période, la nouvelle version de GCC (*5*) a changé sa politique d'allocation de registres pouvant mener la fonction `__strcspn_g` à provoquer une erreur de segmentation (*segmentation fault*).

En quoi consiste le code. La fonction `__strcspn_g` calcule la longueur de la sous-chaîne de caractères de `__s` qui ne contient aucun caractère de l'ensemble `__reject`. Il n'est pas nécessaire de comprendre le fonctionnement de la fonction pour comprendre le problème d'interface qui va être discuté. Les explications qui suivent donnent tout de même un aperçu du rôle des différentes instructions.

Le bloc contient des instructions dont les arguments sont implicites, en particulier les instructions de manipulation de chaînes de caractères :

— `cld` (ligne 1573, *clear direction flag*) :

```
d = 0;
```

— `lodsb` (ligne 1578, *load string byte*) :

```
%al = *(%esi), %esi += 1 - 2*d;
```

— `repne scasb` (ligne 1583, *repeat while not equal compare string byte*) :

```
while ((%ecx > 0) && (z = %al == *(%edi), %esi += 1 - 2*d, !z));
```

```
1563  # ifdef __PIC__

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  }

1618  # endif
```

FIGURE 2.4 – bits/string.h@glibc_2.19

La fonction fait usage de 6 des 7 registres généraux du x86 :

— `%esi` (`"S"`), `%eax` (`"a"`), `%ecx` (`"c"`) et `%edi` (`"D"`) ont un rôle implicite dans les instructions présentées ci-dessus ;

— `%ebx` (*hard coded*) est utilisé pour sauvegarder la longueur de la chaîne `__reject`, calculée entre les lignes 1574 et 1576 ;

— `%edx` ou `%ebp` (« choix » restant pour `"r"`) est utilisé pour sauvegarder le pointeur de la chaîne `__reject`.

Le registre `%ebx` n'apparaît pas dans l'interface. En effet, le compilateur GCC, en version 4 et antérieur, ne permet pas d'utiliser le registre `%ebx` dans un bloc assembleur lorsque le code est compilé avec l'option `-fPIC` (*position independent code*). Les développeurs contournent le problème en utilisant `%ebx` sans le déclarer dans l'interface, il est « écrit en dur » (*hard coded*) dans le code. Ils prennent toutefois soin de le sauvegarder et de le restaurer « manuellement » dans et depuis la pile à l'aide de `push` et `pop` (lignes 1571 et 1586).

Le problème. La version 5 de GCC a levé l'interdiction d'utiliser `%ebx` dans un contexte *PIC*, le rendant éligible à la contrainte générale `"r"`. Ainsi, le compilateur ne sachant pas qu'`%ebx` est utilisé à son insu, il lui est possible de choisir de l'allouer au jeton `%4`. Or le jeton `%4` est lu à la ligne 1581 après qu'`%ebx` se soit fait écraser à la ligne 1576. *Si* `%4` *et* `%ebx` *représentent le même registre, la valeur du pointeur* `__reject` *sera alors remplacée par la longueur de son contenu avant d'être déréférencée, provoquant à coup sûr une erreur de segmentation (*`segfault`*).*

Type de problème. L'allocation de registres est régie par le langage de contrainte (expliqué en détail en section 3.2.2) utilisé pour lier les expressions C aux opérandes assembleur. Le compilateur a pour objectif de minimiser le nombre de registres alloués ainsi que les déplacements de données pour faire en sorte de respecter les contraintes spécifiées. Lorsque des contraintes sont oubliées, le compilateur peut choisir une allocation de registres « valide », dans le sens où les instructions sont bien formées, mais dont le résultat à l'exécution ne sera pas celui attendu (*bug*). Lorsque le comportement du programme dépend de choix réalisés par le compilateur, nous parlerons de violation de la propriété d'**unicité**. Nous formalisons la condition d'unicité en section 5.1.3.

Pourquoi le code semble fonctionner. Un problème d'unicité survient lorsqu'il existe au moins deux allocations possibles de registres qui ne produisent pas le même comportement. Pour peu que le compilateur privilégie la « bonne » allocation, le programme fonctionnera correctement. La version du compilateur, les options de compilation et le contexte d'appel peuvent influencer le choix de l'allocation de registres. Nous pouvons toutefois nous attendre à ce que l'allocation soit stable si les paramètres ne varient pas. Aussi, un bloc qui « a fonctionné correctement » a de grandes chances que cela continue pour un certain temps.

Nous venons de voir deux exemples de blocs assembleur embarqué dont le code assembleur est correct mais dont le résultat après compilation n'est pas celui attendu. Il s'agit de défauts de l'interface entre le C et l'assembleur qui se manifestent en bugs en fonction du contexte de compilation. À cela s'ajoute la difficulté d'écrire du code assembleur correctement. L'écriture de code bas niveau est en effet une source importante d'erreurs. Aussi allons-nous maintenant nous intéresser aux moyens à notre disposition pour vérifier la correction d'un bloc embarqué dans un code C.

2.3 Difficultés avec les analyseurs de niveau C

L'assembleur embarqué fait partie des aspects du langage C qui ne sont pas ou peu supportés par les outils d'analyse de code existants. Nous allons illustrer l'impact que peut avoir un bloc assembleur sur la tâche de vérification avec trois outils représentant les analyses standards de code : KLEE [CDE08] pour l'exécution symbolique, Frama-C EVA [BBY17] pour l'interprétation abstraite et Frama-C WP [CMPP11] pour la vérification déductive. Ces classes d'analyses seront présentées en section 3.1.1.

À l'heure actuelle, les codes contenant de l'assembleur embarqué sont donc plus ou moins hors de portée de vérification. Il existe pourtant des situations pour lesquelles l'assembleur s'invite dans le code un peu à l'insu du développeur. Regardons cela sur le code de la figure 2.5, extrait d'UDPCast.

```
1074 int prepareForSelect(int *socks, int nr, fd_set *read_set) {
1075     int i;
1076     int maxFd;
1077     FD_ZERO(read_set);
1078     maxFd=-1;
1079     for(i=0; i<nr; i++) {
1080         if(socks[i] == -1)
1081             continue;
1082         FD_SET(socks[i], read_set);
1083         if(socks[i] > maxFd)
1084             maxFd = socks[i];
1085     }
1086     return maxFd;
```

FIGURE 2.5 – `socklib.c`@444de59

D'apparence tout à fait normal, le code utilise à la ligne 1077 la macro `FD_ZERO` qui se révèle être définie dans la libc et contenir de l'assembleur embarqué. L'assembleur embarqué est en effet souvent utilisé de pair avec des macros qui permettent de l'insérer avec plus d'aisance dans le code utilisateur.

Cette macro `FD_ZERO` est définie dans le code de libc (`x86/bits/select.h`@217d45c) de la manière suivante :

```
30 #  define __FD_ZERO_STOS "stosl"
31 # endif
32
33 # define __FD_ZERO(fdsp) \
34   do {                                                                    \
35     int __d0, __d1;                                                       \
36     __asm__ __volatile__ ("cld; rep; " __FD_ZERO_STOS                     \
37                           : "=c" (__d0), "=D" (__d1)                      \
38                           : "a" (0), "0" (sizeof (fd_set)                 \
39                                           / sizeof (__fd_mask)),          \
40                             "1" (&__FDS_BITS (fdsp)[0])                   \
41                           : "memory");                                    \
42   } while (0)
```

La définition des macros manquantes est donnée dans les annexes A.1 à A.3.

En quoi consiste le code. La macro `FD_ZERO` sert à initialiser une structure de type `fd_set` (ensemble de descripteurs de fichiers) en remplissant sa table de zéros (`__FDS_BITS` non montré ici). Une fois encore, le code assembleur utilise les primitives de manipulation de chaînes de caractères dont `stos` qui fait l'action opposée de `lods` vue précédemment :

— `cld` (ligne 36, *clear direction flag*) :

$$d = 0;$$

— `rep stosl` (ligne 36, *repeat store string long*) :

```
while (%ecx > 0) *(%edi) = %eax, %edi += 1 - 2*d;
```

L'interface est définie (correctement) de la ligne 37 à la ligne 41 :

— `%ecx`, le compteur de boucle est déclaré en écriture (`"=c"`) et en lecture (`"0"`). Il est initialisé par l'expression (`sizeof (fd_set) / sizeof (__df_mask)`), soit la taille de la table que divise la taille des éléments. Un ensemble de descripteurs de fichiers comprend 1024 éléments ;

— `%edi`, le pointeur auto-incrémenté est déclaré lui aussi en écriture (`"=D"`) et en lecture (`"1"`) et reçoit en entrée la base de la table (`&__FDS_BITS (fdsp)[0]`) ;

— `%eax`, la donnée est déclarée en lecture (`"a"`) et est mise à `0` ;

— enfin le mot clé `"memory"` permet d'indiquer que la mémoire est manipulée par le bloc. Ici, le contenu pointé par `%edi` est modifié.

Ce que l'on veut vérifier. À minima, lorsqu'une fonction manipule des pointeurs et des structures, il est important de s'assurer de la sûreté de la mémoire (*memory safety*) et de garantir qu'il n'y a pas de débordement de tampons (*buffer overflow*).

Pour des entrées de taille modeste, l'exécution symbolique [CDE08] est capable d'énumérer tous les chemins de la fonction et de vérifier qu'aucun d'entre eux ne provoque d'erreur. Par exemple, dans la fonction `prepareForSelect` définie en figure 2.5, l'exécution symbolique explore 2046 chemins si le nombre d'éléments du

tableau `nr` est borné à 10. En comparaison, il faudrait énumérer environ 10^{30} configurations différentes pour explorer exhaustivement l'espace d'entrée en force brute.

L'interprétation abstraite [Cou96] permet aussi de vérifier ce type de propriétés. En présence de boucles, l'interprétation abstraite passe plus facilement à l'échelle que l'exécution symbolique. L'abstraction par des intervalles est bien adaptée à cet exemple puisqu'il s'agit de vérifier que les valeurs des indices des tableaux `socks` et `read_set` sont bien comprises entre `0` et, respectivement, `nr` et `FD_SETSIZE`.

Plus difficile à mettre en place, la vérification déductive [Dij75] permet une vérification modulaire des fonctions. Moyennant un investissement manuel plus conséquent, elle permet de vérifier les propriétés fonctionnelles du code. Frama-C permet d'annoter les fonctions avec des contrats écrit dans le langage de spécification ACSL [BCF+20]. Il est ainsi possible de décrire le comportement attendu de la fonction `prepareForSelect`, à savoir que chaque descripteur de fichier présent dans `socks` se retrouve dans l'ensemble `read_set` et que chaque descripteur de fichier présent dans ce dernier provient de `socks` :

```
/*@ ensures \forall integer i;
      0 <= i < nr  ==> (socks[i] == -1) || FD_ISSET(socks[i], read_set);
    ensures \forall integer k;
      0 <= k < FD_SETSIZE ==> !FD_ISSET(k, read_set) || \exists integer i;
        0 <= i < nr ==> socks[i] == k;
*/
```

Pour notre exemple, nous utiliserons la vérification déductive pour montrer les mêmes propriétés que l'exécution symbolique et l'interprétation abstraite, ce qui se traduit dans le langage de spécification par la clause `assigns` suivante :

```
/*@ assigns *(__FDS_BITS(read_set) +
                  (0 .. sizeof (fd_set) / sizeof (__fd_mask) - 1));
*/
```

2.3.1 Utilisation de KLEE

L'outil d'analyse symbolique KLEE demande d'appeler la fonction dans un contexte (`main`) qui déclare les variables symboliques. Voici un extrait du fichier utilisé pour tester la fonction `prepareForSelect` à l'aide de KLEE (la version complète de ce fichier est donnée en annexe A.1) :

```
57 #define N 10
58
59 int main (int argc, char *argv[])
60 {
61   int socks[N];
62   int nr;
63   fd_set read_set;
64
65   klee_make_symbolic(&nr, sizeof(nr), "nr");
66   klee_assume((0 < nr) & (nr <= N));
67   klee_make_symbolic(&socks, sizeof(socks), "socks");
68   for (int i = 0; i < N; i += 1)
69     klee_assume((-1 <= socks[i]) & (socks[i] < FD_SETSIZE));
70   prepareForSelect(socks, nr, &read_set);
71   for (int i = 0; i < nr; i += 1)
72     if (socks[i] != -1)
73       klee_assert(FD_ISSET(socks[i], &read_set));
74   return 0;
75 }
```

La taille du tableau `socks` est arbitrairement bornée à `nr` = N = 10 éléments (ligne 66). Le code utilise les primitives `klee_make_symbolic` et `klee_assume` pour configurer l'analyse :

— `nr` $\in [0, \texttt{N}[$;

— $\forall i \in [0, \texttt{N}[, \texttt{socks}[i] \in [-1, \texttt{FD\_SETSIZE}[$.

Malheureusement, KLEE (v2.0) ne supporte pas l'assembleur embarqué et, par conséquent, interrompt son exploration lorsqu'un chemin traverse un bloc d'assembleur. Dans la fonction `prepareForSelect`, la macro `FD_ZERO` précède toutes les autres instructions et KLEE s'arrête donc sur le premier chemin avec le message suivant :

```
KLEE: WARNING ONCE: function "prepareForSelect" has inline asm
KLEE: ERROR: motivation.c:36: inline assembly is unsupported
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 279
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

Contournement. Maintenant que nous savons à quoi sert la macro `FD_ZERO`, nous pouvons essayer de la remplacer manuellement par du code C :

```
#undef FD_ZERO
#define FD_ZERO(fdsp)                                                    \
    for (int i = 0; i < sizeof (fd_set) / sizeof (__fd_mask); i += 1)    \
      __FDS_BITS(fdsp)[i] = 0;
```

Avec cette nouvelle configuration, KLEE est désormais capable d'explorer tous les chemins (2046) sans rencontrer d'erreur en un peu plus de 30 minutes :

```
KLEE: done: total instructions = 448682
KLEE: done: completed paths = 2046
KLEE: done: generated tests = 2046

real        36m53.507s
```

2.3.2 Utilisation de Frama-C

L'outil Frama-C offre un support limité de l'assembleur embarqué. Aussi, la syntaxe ne provoquera pas d'erreur mais les analyses seront dans la majorité des cas incapables de donner une conclusion intéressante.

En l'occurrence, la version *19* de Frama-C (Potassium) lève un avertissement problématique lorsque le bloc assembleur de la fonction `prepareForSelect` est traduite dans le langage intermédiaire CIL :

```
[kernel] motivation.c:43: Warning:
  Clobber list contains "memory" argument.
  Assuming no side effects beyond those mentioned in operands.
```

Ce message signifie que le mot clé `memory` n'est pas supporté et que les analyses considéreront que le bloc assembleur n'a pas d'impact sur la mémoire, ce qui est évidement faux dans notre exemple.

Les versions plus récentes n'ont plus cet avertissement. En effet, Frama-C rajoute des annotations autour du bloc assembleur pour prendre en compte les potentiels effets du bloc assembleur sur la mémoire :

```
/*@ assigns __d0, __d1, *(__FDS_BITS (read_set) + ( .. )); */
```

La clause `assigns` déclare que le bloc est susceptible de modifier les variables `__d0` et `__d1`, ainsi que tous les emplacements mémoire accessibles depuis le pointeur `read_set`. Une clause `assigns` ne dit cependant rien sur comment les valeurs sont modifiées. De plus, l'accès mémoire n'est pas borné (sur-approximé par $]-\infty, +\infty[$). Cette approche est malheureusement le seul moyen de prendre en compte les informations présentes dans l'interface de façon correcte sans être en mesure de comprendre les instructions assembleur.

EVA. Le greffon EVA est une analyse globale par interprétation abstraite qui a besoin d'un contexte (`main`). Voici un extrait du fichier utilisé pour tester EVA (la version complète est donnée en annexe A.2) :

```
58
59 int main (int argc, char *argv[])
60 {
61   int socks[N];
62   int nr;
63   fd_set read_set;
64
65   nr = Frama_C_interval(1, N);
66   for (int i = 0; i < N; i += 1)
67     socks[i] = Frama_C_interval(-1, FD_SETSIZE - 1);
68   prepareForSelect(socks, nr, &read_set);
69   return 0;
70 }
```

La primitive `Frama_C_interval` permet ici d'initialiser de façon non-déterministe les variables `nr` et `socks`. Sans plus d'information sur le bloc assembleur et son rôle d'initialisation, EVA émet plusieurs (fausses) alarmes sur le fait que le contenu de `read_set` est utilisé non initialisé :

```
[eva:alarm] motivation.c:49: Warning:
  accessing uninitialized left-value.
  assert
  \initialized(&read_set->__fds_bits[*(socks + i) /
                                     (8 * (int)sizeof(__fd_mask))]);
```

En outre, les accès mémoire de la macro `__FD_ZERO` étant cachés dans le bloc assembleur, l'analyse serait incapable de lever une alarme en cas de débordement de tableau.

WP. Le greffon WP permet une analyse modulaire de la fonction à l'aide d'annotations manuelles ajoutées aux signatures de fonctions (contrats) ou, au besoin, à divers endroits du code, comme les boucles (*loop invariants*).

Ici, nous voulons que la fonction `prepareForSelect` respecte le contrat suivant (la version complète est donnée en annexe A.3) :

```
38 /*@
39   requires \valid_read(socks + (0 .. nr - 1));
40   requires \valid(read_set);
41   requires \forall integer i, j;
42     0 <= i < nr && 0 <= j < sizeof (fd_set) / sizeof (__fd_mask)
43     ==> socks + i != __FDS_BITS(read_set) + j;
44   requires 0 < nr <= FD_SETSIZE ;
45   requires \forall integer i;
46     0 <= i < nr ==> -1 <= socks[i] < FD_SETSIZE;
47   assigns *(__FDS_BITS(read_set)
48             + (0 .. sizeof (fd_set) / sizeof (__fd_mask) - 1));
49 */
```

Les clauses `requires` (lignes 39 à 46) posent toutes les conditions que doit respecter le contexte d'appel. La clause `assigns` est notre objectif de preuve. Il est alors impossible pour l'outil de réussir à prouver que toutes les écritures ont bien lieu au sein des bornes du tableau en partant de l'hypothèse sur-approximante que la macro `FD_ZERO` peut d'écrire à n'importe quel indice, qu'il soit valide ou invalide.

Contournement. Il est possible d'expérimenter avec une traduction manuelle pour vérifier que le bloc assembleur est le seul obstacle à la vérification automatique de cette fonction. Pour EVA, il suffit simplement de remplacer *manuellement* la macro par une boucle naturelle écrite en C (annexe A.2). Pour WP, cela demande un effort supplémentaire car les boucles doivent être annotées pour que l'outil puisse arriver à une conclusion (annexe A.3).

2.4 Ce que propose cette thèse

Nous venons de voir que l'utilisation d'assembleur embarqué pose deux problèmes majeurs :

1. sa syntaxe est compliquée et sujette aux erreurs. Ces erreurs, à l'instar des comportements indéfinis du C, ne provoquent pas toujours directement un bug et sont donc d'autant plus difficiles à comprendre et à identifier ("*bug latent*") ;
2. les outils d'analyse de niveau C ne supportent pas ou très peu la syntaxe de l'assembleur embarqué, ce qui empêche de mener à bien les tâches de vérification sur des codes de bas niveau.

Nous proposons donc deux approches complémentaires pour régler chacun des problèmes :

1. une passe de vérification de la conformité entre les déclarations de l'interface niveau C et les instructions assembleur réelles (chapitre 5) ;

2. une traduction de la sémantique des dites instructions dans la syntaxe du C (chapitre 6).

Ces deux points reposent par ailleurs sur la capacité de comprendre le rôle des instructions assembleur qui sont données simplement sous la forme de chaînes de caractères formatées et incomplètes. Pour cela, nous définissons une représentation intermédiaire et la sémantique opérationnelle associée (chapitre 4), et proposons une méthode d'extraction automatique de cette représentation.

Les méthodes d'extraction, d'analyse et de traduction proposées dans cette thèse ont été implémentées dans le prototype TInA et testées sur un large ensemble de codes assembleur provenant de codes sources publics de la distribution Debian *Jessie*.

Chapitre 3

Contexte et connaissances essentielles

Sommaire

Ce chapitre permet d'introduire les notions et connaissances générales du contexte de la thèse.

Nous présentons dans un premier temps les méthodes formelles et la vérification de programme (section 3.1.1) avant de discuter des difficultés liées à l'application de ces méthodes sur du code de bas niveau (section 3.1.2). Nous détaillons ensuite l'analyse de flot de données (section 3.1.3) que nous utilisons à plusieurs reprises dans la thèse pour vérifier la conformité de l'interface (section 5.2.1) et simplifier le code produit lors de la traduction vers le C (section 6.3.4).

Nous présentons enfin l'assembleur embarqué en détaillant les différences entre les deux syntaxes supportées par les compilateurs contemporains : la syntaxe Microsoft de plus haut niveau (section 3.2.1) et la syntaxe GNU sur laquelle porte cette thèse. Nous finirons sur les résultats de notre étude empirique réalisée sur l'ensemble des blocs assembleur rencontrés dans les sources des paquets de la distribution Debian *Jessie* et qui vise à donner les caractéristiques générales de l'utilisation de l'assembleur embarqué (section 3.2.3).

3.1 Méthodes formelles et analyse de programmes

3.1.1 Présentation générale

Les méthodes formelles visent à raisonner, sur la base des mathématiques, sur des propriétés de programmes. On retrouve parmi les précurseurs des méthodes formelles Floyd [Flo67], Hoare [Hoa69], Dijkstra [Dij76], Cousot [CC77] ou encore Clark [CE81]. Les approches peuvent différer sensiblement mais toutes reposent sur trois éléments clés :

— la sémantique formelle $\mathbb{M}$ des comportements possibles du programme (généralement une sémantique opérationnelle) ;

— la spécification ϕ des comportements correctes (généralement une formule logique) ;

— une (semi-)procédure de décision pour vérifier que les comportements possibles du programme sont corrects, noté $\mathbb{M} \vDash \phi$.

Au vu de la complexité des programmes réels, le problème auquel s'attaquent les méthodes formelles est généralement indécidable. En d'autres termes, il n'est pas possible d'écrire un algorithme capable de répondre avec certitude à la question « le comportement du programme est-il toujours correct ? » pour tout programme.

Les années 2000 marquent, pourtant, le début de formidables avancées dans le domaine avec l'arrivée de techniques s'appliquant avec succès sur du code réel [HJMS03, VPK04, BCLR04, GKS05, CCF+05, KT14, KKP+15]. Les outils développés font « de leur mieux » pour répondre aux questions posées en se servant de **sur-approximation**s ou de **sous-approximation**s.

Sur-approximation. Les sur-approximations sont utilisées lorsque l'on souhaite prouver avec certitude l'absence de comportements indésirables dans un programme.

L'idée est de représenter les comportements du programme sous une forme plus simple à calculer que la forme réelle. L'approximation doit garantir que la forme simplifiée inclut tous les comportements réels du programme. Dans l'exemple de la figure 3.1, le cercle **S'** est bien une sur-approximation de la patatoïde **S**.

Puisqu'il n'y a pas d'intersection avec la zone délimitée par la propriété **P1**, il est possible de conclure que **S'**, et par conséquent **S**, vérifient **P1**. En revanche, il y a une intersection entre la zone délimitée par la propriété **P2** et **S'**. Dans ce cas, il n'est pas possible de conclure quoi que ce soit sur **S**. Il s'agit alors d'un **faux positif**, c'est à dire qu'une violation est détectée sur l'approximation alors que le système réel vérifie la propriété.

La vérification automatique de programme telle que l'**interprétation abstraite** [CCF+05] utilise les sur-approximations pour prouver plus facilement les propriétés étudiées. En pratique, la vérification déductive assistée telle que le **calcul de plus faible précondition** (*Weakest precondition calculus*) [CdSSPT14] fait aussi usage de sur-approximation et demande des hypothèses conservatrices sur les *contrats* et *invariants de boucle* (Calcul de plus faible précondition, page 30).

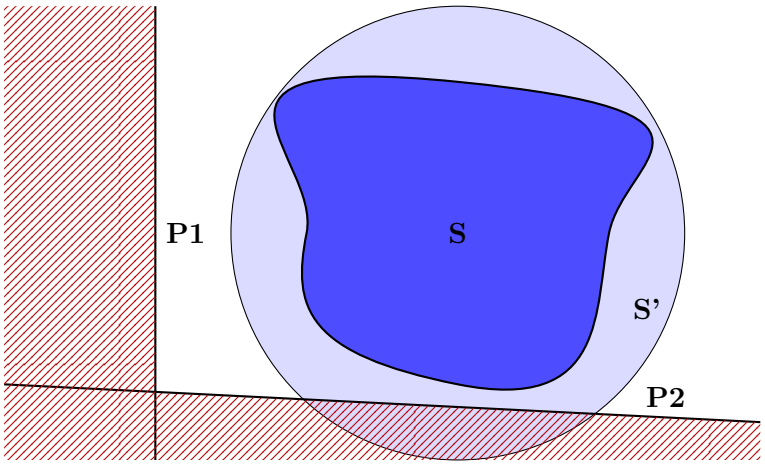


FIGURE 3.1 – Exemple de sur-approximation

Sous-approximation. À l'inverse, les sous-approximations sont utilisées pour montrer avec certitude la présence de comportements indésirables dans un programme.

L'approximation doit ici garantir que la forme simplifiée est entièrement incluse dans les comportements réels du programme. Dans l'exemple de la figure 3.2, le losange **S'** est bien une sous-approximation de la patatoïde **S**.

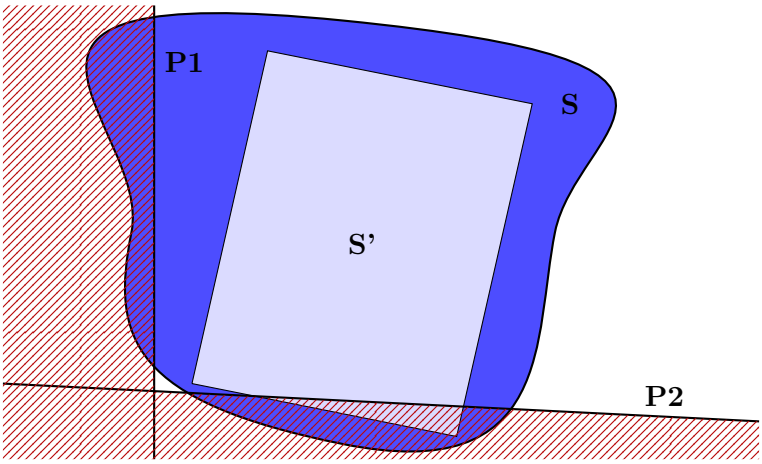


FIGURE 3.2 – Exemple de sous-approximation

Sachant qu'il y a une intersection avec la zone délimitée par la propriété **P2**, il est possible de conclure que **S'**, et par conséquent **S**, violent **P2**. En revanche, l'absence

d'intersection entre la zone délimitée par **P1** et **S'** ne signifie pas que **S** respecte **P1**. Il s'agit dans ce cas d'un **faux négatif**, c'est à dire qu'une violation n'est pas détectée par l'abstraction alors que le système réel ne respecte pas la propriété.

On retrouve les techniques de sous-approximation dans les analyses de détection de bugs et de vulnérabilités telles que l'**exécution symbolique** [CDE08, BGM13] ou la vérification de modèle bornée (*bounded model checking*) [CKOS04].

Lien avec cette thèse. Nous cherchons dans cette thèse à étendre les techniques de vérification formelle de programmes C au cas de code mixte contenant de l'assembleur embarqué qui n'est actuellement pas géré par les méthodes usuelles se plaçant au niveau du C-ANSI. Nous proposons notamment une traduction de l'assembleur embarqué vers le C, validée et optimisée pour faciliter la vérification par les outils de vérification niveau C existants. Nous montrerons l'efficacité de notre approche de traduction (section 6.5) sur trois des techniques les plus employées :

Interprétation abstraite L'interprétation abstraite est une théorie d'approximation de la sémantique des programmes introduite par Cousot et Cousot [CC77]. Elle sacrifie une part plus ou moins importante de précision pour gagner en calculabilité. La sémantique abstraite peut jouer sur deux niveaux pour approcher la sémantique concrète :

- la représentation des données en omettant des informations non essentielles – par exemple, ne garder que l'information sur l'initialisation des variables, sans tenir compte de la valeur ;
- la réduction du nombre d'états en agrégeant plusieurs états ensemble – par exemple, ne garder qu'un seul état abstrait par point de programme.

L'abstraction choisie dépendra de la propriété que l'on cherche à démontrer. Dans la pratique, l'interprétation abstraite est un processus itératif sur le graphe de flot de contrôle qui évalue la valeur des variables dans un domaine abstrait particulier pouvant représenter des ensembles de valeurs (énumération, intervalles, congruentes, etc.) ou des relations (égalités de valeurs, équations linéaires, etc.). Chaque domaine abstrait définit un treillis de valeur (ensemble d'éléments partiellement ordonnés qui admet une borne supérieure et inférieure) et la sémantique des différents opérateurs sur ce treillis.

Calcul de plus faible précondition La vérification déductive permet de vérifier un contrat sur une portion de code, typiquement une fonction (par exemple, que la fonction `sort` renvoie un tableau trié). Le contrat se compose d'une précondition et d'une postcondition, l'objectif étant de montrer que la précondition est suffisante pour garantir la postcondition. Pour ce faire, le calcul de plus faible précondition raisonne en arrière pour déterminer quelle est la précondition nécessaire pour que l'exécution du code garantisse la postcondition, puis vérifie que la précondition du contrat implique la précondition minimale inférée. L'analyse génère des obligations de preuve en logique du 1^{er} ordre, pouvant être résolues à l'aide de solveurs de contraintes SMT [LQ08], ou en logique du 2^{nd} ordre, nécessitant un assistant de preuve [NPW02, PM12]. Il

est généralement nécessaire d'annoter manuellement les cycles du graphe de flot de contrôle par un résumé logique. Ces annotations appelées invariants de boucle sur-approximent usuellement le comportement du corps de la boucle.

Exécution symbolique L'exécution symbolique permet d'obtenir la relation entre les données d'entrée (arguments de fonction, entrées utilisateur, etc.) et un chemin emprunté par le programme. L'analyse permet de calculer un prédicat de chemin (sous la forme d'une formule logique sans quantificateur) qui doit être satisfait afin que ce chemin soit emprunté. L'exécution symbolique est notamment utilisée pour la génération efficace de tests de couverture de programmes [CS13]. Les analyseurs utilisent les solveurs de contraintes SMT pour vérifier la satisfaisabilité du prédicat de chemin et générer un jeu d'entrées concrètes permettant d'exécuter ce chemin. L'exécution symbolique repose fondamentalement sur l'énumération des chemins d'exécution du programme. En pratique, cette méthode ne sera donc pas toujours à même de compléter un parcours exhaustif de tous ces chemins d'exécution, notamment en présence de boucles non bornées.

3.1.2 Influence du langage de programmation

Les principes généraux des différentes analyses présentées en section 3.1.1 s'appliquent quel que soit le langage étudié. En pratique toutefois, les techniques sont dépendantes des structures du langage et on préférera analyser un code au niveau source plutôt qu'au niveau binaire. En effet, les propriétés que l'on souhaite vérifier sont souvent plus simples à exprimer dans un langage de haut niveau, et parallèlement, ces langages offrent souvent plus de garanties. Nous présentons ici deux différences qui impactent sensiblement le processus de vérification.

Gestion de la mémoire et typage. Les vérifications sur la gestion mémoire ne sont pas abordées de la même façon en Java, en C ou en assembleur. Dans le premier cas, il s'agira de vérifier que le pointeur n'est pas nul alors qu'il faudra s'assurer de la bonne allocation, initialisation et déallocation des blocs mémoire en C. En Java, la taille des objets est connue à l'exécution, évitant par exemple de faire un débordement de tableau (le programme lève une exception). En C, les informations de types peuvent aider à vérifier statiquement l'absence de débordement mémoire. Par ailleurs, des pointeurs de types incompatibles ne peuvent pas s'aliaser en Java et il en est de même en C lorsque la règle d'aliasing strict (*strict aliasing*) est appliquée. En assembleur et en binaire, il n'existe pas nativement de notion de types ni de séparation de mémoires. Aussi, une imprécision dans l'adresse d'une écriture ou d'une lecture en mémoire peut impacter des éléments complètement indépendants (par exemple, les variables locales dans la pile). Pour pallier ce problème, il existe des techniques pour tenter de retrouver ou de créer des types à partir des opérations réalisées dans le code [LAB11, RKS16].

Reconstruction du graphe de flot de contrôle. Le graphe de flot de contrôle d'un langage source est en règle générale facilement approximé par sa structure : fonctions, boucles, etc. Le code binaire présente en revanche un défi de taille pour les analyseurs [BR10, BHV11, MM16]. En effet, la macro-structure du code est perdue lors de la compilation. En absence d'éléments structurants, la découverte du graphe de flot de contrôle devient particulièrement difficile en présence de « sauts dynamiques » (branchements dont la cible de saut dépend de valeurs calculées à l'exécution). Les analyseurs de code binaire souffrent ainsi d'un problème d'amorce : les analyses ont besoin d'une bonne approximation du graphe de flot de contrôle qui a lui-même besoin des résultats des analyses pour être reconstruit, la perte de précision d'un côté impactant négativement la précision de l'autre. Ainsi, dans le cadre de l'interprétation abstraite, la sur-approximation d'une cible de saut peut entraîner l'interprétation d'une partie du programme normalement inatteignable qui pourra elle-même dégrader la précision de l'analyse de valeurs. Le problème est d'autant plus important pour le jeu d'instructions `x86` dont l'encodage est très dense (n'importe quelle suite d'octets ressemble à une instruction valide) et les contraintes d'alignement sont faibles (n'importe quelle adresse est une cible de saut valide).

Les analyseurs de niveau source sont plus matures et rencontrent moins de difficultés à être utilisés. Cependant, le code source n'est pas toujours disponible (code propriétaire, code hérité, etc.) et les compilateurs ne sont pas forcément fiables. Ces raisons [BR10] ont motivé la communauté scientifique à étudier la vérification de code binaire, avec le développement de plusieurs plateformes d'analyse de code [BJAS11, DB15, SWS+16].

3.1.3 Analyse de flot de données

Nous utilisons l'analyse de flot de données à plusieurs reprises pour la vérification de la conformité à l'interface (section 5.2.1) ainsi que dans les passes d'optimisation de la traduction (section 6.3.4). Cette section sert à introduire les notions principales utiles à ces analyses.

L'analyse de flot de données joue un rôle important dans le domaine de l'analyse statique. Elle est largement utilisée dans les compilateurs [Kil73] à des fins d'optimisation. Elle consiste à calculer des faits (*dataflow fact*) sur le programme en chacun des nœuds de son graphe de flot de contrôle.

Le flot de données peut être vue comme un système d'équations qui relie les nœuds du graphe de flot de contrôle et dont la forme générale est :

$$\begin{aligned} Out_{\mathtt{n}} &= \mathtt{Transfer}_n(In_{\mathtt{n}}) \\ In_{\mathtt{n}} &= \mathop{\mathtt{Join}}_{\mathtt{n}' \in incomings(\mathtt{n})}(Out_{\mathtt{n}'}) \end{aligned}$$

La fonction $\texttt{Transfer}_{\texttt{n}}$ définit la transformation qu'opère le noeud `n` sur le fait $In_{\texttt{n}}$ donné en entrée, ce dernier étant calculé comme l'agglomération `Join` de tous les faits $Out_{\texttt{n}'}$ des noeuds `n'` arrivants sur `n`.

Les équations sont résolues de manière itérative jusqu'à ce que le calcul converge. L'état stable du système est appelé **point fixe**. Dans la pratique, la convergence vers le point fixe est garantie grâce à deux hypothèses :

— le domaine de valeurs des faits est assimilable à un treillis de hauteur finie ;

— les fonctions `Transfer` et `Join` sont monotones sur ce treillis.

Ainsi, la sortie ne pouvant que croître ou rester la même et, la croissance étant limitée par la hauteur du treillis, le système se stabilise nécessairement dans un état pour lequel la sortie est la même d'une itération à l'autre.

L'analyse est dite « en avant » (*forward*) lorsque les équations d'un nœud dépendent uniquement de celles de ses prédécesseurs. À l'inverse, elle est dite « en arrière » (*backward*) lorsque les équations d'un nœud dépendent de celles de ses successeurs.

L'algorithme de résolution de l'analyse de flot de données à l'aide d'une liste de travail (*worklist*) est illustré dans le pseudo-code 1. L'algorithme est paramétré par le type abstrait des faits calculés (`data`) et par les deux opérations `Transfer` et `Join`. Le sens de l'analyse (*backward* ou *forward*) est configuré par les fonctions `FirstNode` et `NextEdges` qui peuvent prendre respectivement les valeurs `entryNode` et `forwardEdges` ou `exitNode` et `backwardEdges`. La fonction `Init` donne l'approximation initiale des valeurs de chaque nœud du graphe de flot de contrôle.

Pseudo-code 1: Algorithme générique de l'analyse de flots de données

```
require data : type
require FirstNode : graph → node
require NextEdges : node → edge set
require Init : node → data
require Transfer : edge × data → data
require Join : data × data → data

foreach node ∈ cfg do
  | result[node] ⟵ Init(node)
worklist ⟵ NextEdges(FirstNode(cfg))
while worklist ≠ ∅ do
  | edge ⟵ Pop(worklist)
  | data ⟵ Transfer(edge, result[edge.src])
  | data ⟵ Join(data, result[edge.dst])
  | if data ≠ result[edge.dst] then
  |   | result[edge.dst] ⟵ data
  |   | worklist ⟵ worklist ∪ NextEdges(edge.dst)
end
```

Nous donnons maintenant deux exemples d'analyse de flot de données :

— l'analyse d'accessibilité (*reachability analysis*) qui permet de calculer si un noeud du graphe peut être atteint ;

— l'analyse de vivacité (*liveness analysis*) qui permet de calculer si une valeur est utilisée par la suite.

Accessibilité. Le calcul d'accessibilité est utilisé pour retirer les nœuds qui ne sont pas atteignables depuis un nœud donné (point d'entrée) et sert donc à l'élimination de code « mort » (*dead code*). Il se calcule très simplement avec l'analyse de flot de données en avant : le point d'entrée est déclaré atteignable, un nœud est atteignable si au moins un de ses prédécesseurs est atteignable.

Nous donnons dans l'instance 1 la définition des fonctions du pseudo-code 1 pour cette analyse.

Flot de données 1: Calcul d'accessibilité

```
type data : bool
define FirstNode(cfg) = cfg.entryNode
define NextEdges(node) = node.forwardEdges
define Init(node) = node.isEntry
define Transfer(edge, data) = data
define Join(data, data') = data ∨ data'
```

Vivacité. L'analyse de vivacité permet de déterminer si une variable est utilisée par la suite. Il est par exemple possible d'éliminer les affectations aux variables qui ne sont plus jamais lues. La vivacité se calcule facilement à l'aide d'une analyse de flot de données en arrière en partant des points de sortie du graphe (par exemple un `return`). Pour un nœud donné, une variable est vivante, soit parce qu'elle est utilisée par le nœud, soit parce qu'elle est vivante pour au moins un des prédécesseurs du nœud et qu'elle n'est pas mise à jour dans ce dernier.

Le paramétrage de l'analyse de flot de données dépend de la structure du langage utilisé (instructions et expressions). Un exemple de configuration est donné en section 5.2.1 (algorithme 3).

3.2 Assembleur embarqué dans du C

La grande majorité des compilateurs C propose l'extension de l'assembleur embarqué. Cette extension se décline en deux familles : l'assembleur Microsoft, proposé par VisualStudio et supporté par ICC, et l'assembleur GNU proposé par GCC et largement adopté par Clang et ICC.

Bien que ces deux familles remplissent un rôle similaire, leurs choix de conception sont radicalement différents. Ainsi, après avoir présenté succinctement les caractéristiques de l'assembleur Microsoft, nous n'aborderons dans la suite du document que les problématiques liées à l'utilisation de l'assembleur GNU.

3.2.1 Syntaxe de Microsoft

La syntaxe proposée par Microsoft est certainement la plus facile à utiliser. Le mot clé `__asm` (imposé par la norme C) peut ainsi être suivi d'une instruction assembleur ou d'un groupe entouré d'accolades.

Le bloc donné en figure 3.3 présente la variante Microsoft de la fonction en figure 2.2 :

```
1 int mid_pred (int a, int b, int c)
2 {
3   __asm
4   {
5     mov   eax, b       // load argument b in eax
6     mov   edx, a       // load argument a in edx
7     mov   ecx, eax
8     mov   esi, c       // load argument c in esi
9     cmp   edx, ecx
10    cmovg eax, edx
11    cmovg edx, ecx
12    cmp   edx, esi
13    cmovl edx, esi
14    cmp   eax, edx
15    cmovg eax, edx     // return is already in eax
16  }
17 }
```

FIGURE 3.3 – Exemple de bloc assembleur dans la syntaxe Microsoft

Outre le fait que le dialecte assembleur Intel soit utilisé à la place de celui d'AT&T [1], des différences notables apparaissent :

— les instructions assembleur sont écrites telles quelles sans passer par une chaîne de caractères ;

— il n'y a pas d'interface pour lier l'assembleur au C. À la place, les variables C sont directement insérées dans le code assembleur (lignes 5, 6 et 8).

Cela est possible car, contrairement à ce que nous verrons pour GNU, VisualStudio et ICC ont connaissance d'une partie des instructions assembleur utilisées. La syntaxe Microsoft se rapproche donc de l'usage de primitives (builtin). Le compilateur est ainsi informé que les registres `%eax`, `%edx`, `%ecx` et `%esi` sont utilisés et modifiés sans qu'il soit nécessaire de le spécifier. Il sait également que, le registre `%eax` étant mis à jour, la fonction a une valeur de retour. Une telle approche résout nombre de problèmes que nous allons voir par la suite. Cependant, le développeur est en charge de choisir les registres utilisés, ce qui retire au compilateur un pouvoir de décision et, par conséquent, des opportunités d'optimisation. Le coût de développement du compilateur est nécessairement plus important et cette syntaxe est actuellement limitée à un sous-ensemble du jeu d'instructions x86 dans sa version 32 bit. *L'assembleur embarqué de Microsoft est donc un assembleur de plus haut niveau, permettant de*

1. Les opérandes source et destination sont inversés entre les deux dialectes.

lier automatiquement le C et l'assembleur mais dont la gestion native du compilateur reste limitée.

3.2.2 Syntaxe de GNU

La syntaxe proposée par GNU a déjà été rencontrée dans le chapitre 2. Elle s'applique à toutes les architectures car les instructions sont données sous forme de chaînes de caractères qui seront directement transmises à l'émission de l'assembleur cible. Cette syntaxe impose cependant d'écrire l'interface du bloc pour que le compilateur puisse faire le lien avec le reste du code. Cette interface peut notamment laisser au compilateur le soin de choisir les registres.

```
⟨statement⟩  ::= 'asm' [ 'volatile' ] '(' ⟨template :string⟩ [ ⟨interface⟩ ] ')'
 ⟨interface⟩  ::= ':' [ ⟨outputs⟩ ] ':' [ ⟨inputs⟩ ] ':' [ ⟨clobbers⟩ ]

   ⟨outputs⟩  ::= ⟨output⟩ [ ',' ⟨outputs⟩ ]
    ⟨inputs⟩  ::= ⟨input⟩ [ ',' ⟨inputs⟩ ]
  ⟨clobbers⟩  ::= ⟨clobber :string⟩ [ ',' ⟨clobbers⟩ ]

   ⟨output⟩  ::= [ '[' ⟨identifier⟩ ']' ] ⟨constraint :string⟩ '(' ⟨Clvalue⟩ ')'
    ⟨input⟩  ::= [ '[' ⟨identifier⟩ ']' ] ⟨constraint :string⟩ '(' ⟨Cexpression⟩ ')'
```

FIGURE 3.4 – Syntaxe concrète de l'assembleur embarqué étendu

Aperçu général La figure 3.4 montre la grammaire du langage pour embarquer un bloc assembleur dans le C. Un bloc peut être basique (**basic**) s'il ne possède pas d'interface. Dans ce cas, il ne doit pas interagir avec l'environnement local C. À l'inverse, il est étendu (**extended**) lorsqu'il est enrichi d'une interface. Nous ne considérerons que le second cas : un bloc assembleur peut être vu comme « une série d'instructions de bas niveau qui transforme un ensemble d'entrées en un ensemble de sorties » [GCC20]. L'interface a le rôle de lier des expressions C et des opérandes assembleur déclarés en tant qu'entrées initialisées (*input*) ou sorties (*output*). L'interface peut également déclarer un ensemble d'emplacements, registres ou mémoire, qui peuvent être modifiés par le bloc mais dont la valeur n'est pas un résultat (calcul intermédiaire, effet de bord, etc.).

Un bloc assembleur peut enfin être qualifié par les mots-clés `volatile`, `inline` ou `goto`. Ces marqueurs ne sont pas pertinents dans le cadre de cette thèse et ne seront pas détaillés.

Pour lier les expressions C et les opérandes assembleur, l'interface utilise un langage de spécification basé sur la notion de patrons (*templates*), de *clobbers* et de langage de contraintes. Nous allons maintenant détailler ces différents éléments qui nous serons utiles par la suite (section 4.3.3).

Patron et jetons. Le code assembleur est donné sous la forme d'une chaîne formatée de caractères (*formatted* `string`) similaire à celle de la fonction `printf`. Elle peut contenir des jetons (*token*) qui seront substitués par le compilateur par un

opérande assembleur. Les jetons commencent par le symbole '%', suivi d'un modificateur (*modifier*) optionnel et d'une référence à une des entrées ou des sorties de l'interface. La référence peut se faire par nom symbolique (identifiant entre crochets) ou par position (le numéro de l'élément dans liste des déclaration). Le compilateur traite la chaîne de caractères en substituant les jetons en fonction des modificateurs utilisés [2] et imprime le résultat dans le fichier assembleur de sortie.

Clobbers. Les *clobbers* sont des noms de registre dont la valeur peut être modifiée mais qui ne sont pas désignés comme des sorties du bloc. L'ensemble des registres *clobbers* est disjoint de l'ensemble des registres utilisés par les entrées et les sorties du bloc (absence d'*aliasing*). Le mot clé `"cc"` identifie, lorsqu'il existe sur l'architecture cible, le registre contenant les indicateurs de conditions (*conditional flags*). Le mot clé `"memory"` informe le compilateur que la mémoire dans son ensemble peut être lue et écrite arbitrairement. Ce mot clé force le compilateur à rafraîchir l'ensemble des variables et des blocs mémoires qu'il conservait en cache (registres, tampon temporaires, etc.).

Contraintes. Les contraintes permettent de décrire l'ensemble des opérandes valides pour la substitution d'un jeton. Un opérande peut être de trois types : une valeur immédiate, un registre, ou un emplacement mémoire. Dans ce langage, chaque caractère représente une classe d'opérandes assembleur. La figure 3.5 donne une vue d'ensemble de ces classes de contraintes atomiques définies pour le jeu d'instruction x86. Aussi, par exemple, choisir la lettre `a` ajoute le registre `%eax` aux opérandes assembleur valides pour la substitution de ce jeton. La lettre `m` ajoute quant à elle les opérandes de la forme d'un accès mémoire (par exemple `4(%edi)`).

$$\texttt{a} = \{\texttt{\%eax}\} \qquad \texttt{b} = \{\texttt{\%ebx}\} \qquad \texttt{c} = \{\texttt{\%ecx}\}$$
$$\texttt{d} = \{\texttt{\%edx}\} \qquad \texttt{S} = \{\texttt{\%esi}\} \qquad \texttt{D} = \{\texttt{\%edi}\}$$

$$\texttt{U} = \texttt{a} \cup \texttt{c} \cup \texttt{d} \qquad \texttt{q} = \texttt{Q} = \texttt{a} \cup \texttt{b} \cup \texttt{c} \cup \texttt{d}$$
$$\texttt{i} = \texttt{n} = \mathbb{Z} \qquad \texttt{r} = \texttt{R} = \texttt{q} \cup \texttt{S} \cup \texttt{D} \cup \{\texttt{\%ebp}\}$$

$$\texttt{p} = \{r_b + k \times r_i + c \text{ for } r_b \in \texttt{r} \cup \{\texttt{\%esp}\} \cup \{0\} \text{ and } r_i \in \texttt{r} \cup \{0\} \text{ and } k \in \{1, 2, 4, 8\} \text{ and } c \in \texttt{i}\}$$

$$\texttt{m} = \{*p \text{ for } p \in \texttt{p}\} \qquad \texttt{g} = \texttt{i} \cup \texttt{r} \cup \texttt{m}$$

FIGURE 3.5 – Association entre caractères et classes d'opérandes en x86

2. Seul un sous-ensemble est aujourd'hui documenté pour le x86 [McC07].

Ces contraintes atomiques peuvent ensuite être composées pour former différents ensembles d'opérandes. Lister plusieurs contraintes atomiques réalise l'union de leurs opérandes (par exemple, `"rm"` signifie `r` ∪ `m`). Le langage permet d'organiser les contraintes en alternatives séparées par des virgules. Le compilateur doit choisir une alternative pour tout le bloc (tous les jetons doivent posséder le même nombre d'alternatives) avant de sélectionner les opérandes dans les contraintes qui y sont associer. Ainsi, dans le bloc `asm ("movl %1, %0" : "=r,rm" (a) : "g,ri" (b))`, le compilateur a le choix entre `"=r" (a)` ∧ `"g" (b)` et `"=rm" (a)` ∧ `"ri" (b)`, et ne pourra donc pas sélectionner simultanément une opérande de type mémoire pour chacun des jetons.

Il est possible de forcer le compilateur à choisir le même opérande pour deux jetons par une contrainte « d'appariement » (*matching constraint*). Deux jetons sont appariés lorsque la contrainte d'une entrée fait référence à l'identifiant d'un jeton de sortie (de la même façon que les jetons apparaissent dans le patron). Ainsi, dans le bloc `asm ("" : "=r" (r) : "0" (a))`[3], le jeton `%1` est apparié au jeton `%0` et le même registre sera utilisé pour les deux.

Pour finir, certains caractères ont un rôle particulier (*modifier*) et peuvent changer le comportement des contraintes :

— le symbole `'&'` (*early clobber*) informe le compilateur qu'il ne peut pas utiliser le même opérande pour cette sortie et toutes les entrées qui n'y sont pas appariées ;

— le symbole `'%'` déclare que cette entrée est interchangeable avec l'entrée suivante. Ce symbole peut être utilisé lorsque les opérations sont commutatives (addition, multiplication, etc.) ;

— le symbole `'='` est nécessaire pour déclarer les sorties du bloc (écriture seule) ;

— le symbole `'+'` est nécessaire pour déclarer une entrée-sortie (lecture et écriture).

3.2.3 Caractéristiques de l'assembleur étudié

Afin de bien cerner le problème que représente l'usage de l'assembleur embarqué dans C, nous avons pris pour exemple l'assembleur tel qu'il est utilisé dans les sources des paquets qui composent la distribution Debian. Notre objectif est de déterminer le nombre et le type de blocs assembleur susceptibles d'être rencontrés lors de la vérification de code.

Prévalence. Nous avons écrit un programme parcourant le dépôt d'applications de Debian Jessie (8.11) pour l'architecture i386 (x86 32 bit) et marquant les fichiers sources comprenant un ou plusieurs blocs d'assembleur embarqué[4].

3. Bien que le patron soit vide, ce bloc a pour effet d'affecter la valeur de la variable `a` dans la variable `r` sans que le compilateur ne puisse optimiser le code en aval.

4. Il parait raisonnable de penser que la base de code Jessie, aujourd'hui supplantée par des versions plus récentes, n'a que peu évoluée dans son usage de l'assembleur depuis le commencement de la thèse.

Cela nous a permis d'extraire plus de 3000 blocs assembleur répartis sur environ 200 paquets et dont la taille varie d'une seule ligne à plusieurs centaines. Nous détaillons ici 4 projets qui présentent une grande utilisation de l'assembleur embarqué, que ce soit par le nombre de petits blocs comme GMP, ou la complexité des blocs comme ALSA, FFMPEG ou libyuv.

ALSA : librairie de traitement audio. Elle utilise des instructions de traitement de données parallèles (SIMD) pour accélérer son débit de traitement ;

FFMPEG : outils de conversion de formats vidéo. L'assembleur est là aussi utilisé pour accélérer les primitives de traitement des flux vidéo en utilisant tout particulièrement les instructions SIMD ;

GMP : librairie d'arithmétique multi-précisions permettant de s'abstraire des contraintes liées à la taille des entiers machines. Cette librairie utilise l'assembleur embarqué pour augmenter l'efficacité de ses primitives de base manipulant eux les entiers machines ;

libyuv : librairie de conversion de formats de couleur. Une fonction en particulier utilise les instructions SIMD pour calculer efficacement la somme des carrés des différences entre deux vecteurs d'entrées.

Le tableau 4.1 du chapitre 4 donne le détail précis de la composition des blocs au sein des projets pour les architectures x86 et ARM. Plus de 90% des blocs sont de petite taille (moins de 5 instructions) alors que les blocs plus imposants (de 30 à plus d'une centaine d'instructions) se font rares (environ 2%). Nous retrouvons environ 12% de blocs assembleur qui font appel à des instructions de type "système" (qui ne sont pas supportés par nos techniques).

Une étude similaire a été réalisée par Rigger et al. [RMK+18] sur un ensemble de projets déposés sur GitHub. Les auteurs constatent que plus d'un quart des projets mis en avant sur le site contient de l'assembleur embarqué. Eux aussi relèvent l'usage de multiples blocs de petite taille. En revanche, les auteurs de l'étude ont volontairement évité les blocs de plus grande taille. Notre étude vient ainsi compléter leur analyse en rajoutant l'usage de blocs plus complexes que nous jugeons particulièrement intéressants.

L'assembleur embarqué est présent dans de nombreux codes, que ce soit une utilisation volontaire (ALSA, FFMPEG, etc.) ou par héritage (entête de la librairie libc, etc.). Deux catégories se distinguent dans la masse importante de blocs assembleur embarqué rencontrés : les petits blocs présents en trop grande quantité pour être traités manuellement et les blocs de grande taille plus rares et localisés mais aussi plus complexes.

Caractéristiques. L'assembleur embarqué ainsi observé tel qu'il est utilisé dans le monde du logiciel libre présente de bonnes propriétés qui le distinguent de code assembleur généré par un compilateur :

— Absence de donnée mélangée avec les instructions. Les données proviennent de l'environnement C et sont donc plus facilement traçables dans le code assembleur ;

— Graphe de flot de contrôle statiquement connu. En effet, à l'exception d'appel de fonction passé en paramètre, que nous considérerons hors cadre, il n'est jamais question d'un saut dynamique dans les instructions utilisées ;

— Absence de label vers le C (`asm goto`). Le bloc possède donc un unique point d'entrée et un unique point de sortie ;

— Taille des blocs restreinte. Si l'écriture d'une centaine de lignes d'assembleur est une épreuve pour un être humain, l'analyser ne pose aucun problème de passage à l'échelle pour une machine ;

— Présence de boucle naturelle. La majorité des boucles rencontrées dans ces exemples se comporte comme une boucle for.

Il apparaît ainsi que, bien que de bas niveau, l'assembleur embarqué dans le C évite certaines des difficultés clés du binaire ou de l'assembleur non embarqué, ce qui en fait une bonne cible d'analyse.

Chapitre 4

Sémantique de l'assembleur embarqué

Sommaire

Il est naturel de rapprocher le comportement de l'assembleur embarqué de celui de l'assembleur « classique » mais il ne faut cependant pas le résumer à cela. Comme nous avons pu le voir dans la section 2.2, les jetons (*token*) et l'interface jouent un rôle important dans le fonctionnement final du programme.

Dans ce chapitre, nous proposons une représentation intermédiaire unifiée qui servira de base à la fois pour la vérification de la conformité de l'interface (chapitre 5) et pour la traduction vers la syntaxe du C (chapitre 6). La section 4.1 donne une vision d'ensemble des définitions introduites dans ce chapitre.

Nous définissons dans un premier temps, en section 4.2, la représentation intermédiaire et la sémantique opérationnelle de l'assembleur, positionnée entre la sémantique du C [LB08] et la sémantique binaire pure [BJAS11, BHL+11, SWS+16].

Les spécificités de l'assembleur embarqué, le patron et l'interface, sont ensuite intégrées à cette représentation intermédiaire (section 4.3). La sémantique opérationnelle de l'interface est ensuite définie formellement. Cette étape est nécessaire pour faire le lien entre l'environnement C et les instructions assembleur et ainsi pouvoir traduire correctement le comportement du bloc.

Enfin, pour que cette représentation intermédiaire puisse être utilisée en pratique, nous proposons en section 4.4 une approche pour l'extraire automatiquement d'un code source contenant un bloc assembleur embarqué. Notre prototype, nommé TInA, implémente cette méthode. L'évaluation expérimentale de TInA sur le corpus d'exemples décrit en section 3.2.3 montre que cette approche est efficace pour obtenir la représentation intermédiaire des blocs assembleur tels qu'utilisés par les développeurs en milieu applicatif, avec un taux de succès d'environ **85**%.

4.1 Présentation générale

Dans ce chapitre, nous cherchons à proposer un modèle formel de l'assembleur embarqué qui permet, d'une part, la vérification automatique de la conformité à l'interface (chapitre 5) et d'autre part, la réutilisation des outils de vérification niveau C à l'aide d'une traduction automatique (chapitre 6).

Pour arriver à cet objectif, nous avons besoin de la sémantique précise des instructions assembleur qui composent le patron et de la signification exacte des éléments qui composent l'interface (section 3.2.2). Nous introduisons dans ce chapitre un ensemble de définitions formelles pour représenter ces éléments.

Notations. Dans la suite du document, nous utiliserons la notation `n` : `t` pour désigner un objet `n` de type `t`. Les symboles standard × et ↦ représentent respectivement le produit cartésien [1] et une fonction. L'application d'arguments à une fonction passe par des parenthèses.

La figure 4.1 offre une vue d'ensemble de notre formalisation ainsi que les types utiles. Nous allons maintenant présenter succinctement le rôle que ces types jouent dans notre formalisme.

Notions de base. Un bloc assembleur embarqué contient un patron $\texttt{C}^{\diamond}$: $\texttt{asm}^{\diamond}$, c'est-à-dire une représentation textuelle des instructions assembleur qui peut contenir des jetons `t` : `token` (identifiant syntaxique `%0`, `%1`, etc.). L'interface de la syntaxe concrète (figure 4.1, boîte « *GNU syntax* ») permet au compilateur de choisir une affectation de jetons (*token assignment*) `T` : `token` ↦ `expr` qui associe à chaque jeton `t` un opérande assembleur comme un registre par exemple. Il est alors possible de substituer syntaxiquement les jetons `t` par les opérandes assembleur qui leur sont associés (figure 4.1, flèche de gauche « *instanciate*) pour obtenir un code assembleur « classique » $\texttt{C}^{\alpha}$: $\texttt{asm}^{\alpha}$.

1. Le symbole × est également utilisé dans les expressions pour représenter la multiplication.

Les techniques existantes d'extraction de sémantique de niveau binaire (figure 4.1, boîte « *State of the art* »), telles que celles proposées dans la plateforme BINSEC permettent d'extraire une représentation intermédiaire $\mathtt{C}^{\iota} : \mathtt{asm}^{\iota}$ adaptée aux techniques de vérification automatique de code (section 4.2.2).

Spécificités de l'assembleur embarqué. Pour analyser un morceau d'assembleur embarqué il est nécessaire de comprendre comment le compilateur interprète l'interface concrète pour, d'une part, instancier le patron $\mathtt{C}^{\diamond}$ en choisissant une affectation de jetons $\mathtt{T}$ et, d'autre part, lier les entrées et les sorties du bloc assembleur aux variables de l'environnement C.

Nous définissons en section 4.3.3 une modélisation de l'interface concrète dans une représentation formelle $\mathtt{I}^{\diamond}$: `interface` (figure 4.1, boîte « `interface` »). En particulier, l'interface $\mathtt{I}^{\diamond}$ contient l'ensemble $\mathtt{S}^{\mathtt{T}}$ de toutes les affectations de jetons $\mathtt{T}_n$ valides pour ce bloc assembleur.

L'interface $\mathtt{I}^{\diamond}$ et la représentation $\mathtt{C}^{\iota}$ suffisent à vérifier des propriétés sur l'assembleur embarqué pour une affectation de jetons donnée, mais il est important de raisonner sur les instructions assembleur indépendamment de l'affectation de jetons $\mathtt{T}$ choisie.

Aussi, nous proposons en section 4.4 une méthode qui permet d'extraire automatiquement une représentation intermédiaire augmentée $\mathtt{C}^{\blacklozenge}$: $\mathtt{asm}^{\blacklozenge}$ qui prend en compte le rôle des jetons (figure 4.1, boîte « $\mathtt{asm}^{\blacklozenge}$ »). La présence des jetons dans notre représentation intermédiaire est nécessaire pour raisonner correctement sur la conformité à l'interface (chapitre 5).

Au final, la formalisation du patron $\mathtt{C}^{\blacklozenge}$ et de l'interface $\mathtt{I}^{\diamond}$ nous permet de définir la sémantique opérationnelle d'un morceau d'assembleur embarqué dans du C, autorisant ainsi la traduction automatique de la représentation intermédiaire $\mathtt{C}^{\blacklozenge}$ vers le langage C.

4.2 Représentation et sémantique de l'assembleur

Cette section introduit notre formalisme pour un morceau de code assembleur (*assembly chunk*). Notre langage intermédiaire $\mathtt{asm}^{\iota}$ étend les langages intermédiaires utilisés par les outils de code binaire [BJAS11, BHL+11, SWS+16] en y intégrant des notions liées au modèle mémoire du C [LB08], en s'inspirant des travaux de Djoudi et al. [DB15] et Besson et al. [BBW14].

4.2.1 Représentation intermédiaire de l'assembleur $\mathtt{asm}^{\iota}$

Un morceau de code assembleur $\mathtt{C}^{\alpha} : \mathtt{asm}^{\alpha}$ est une représentation textuelle formée d'étiquettes (*labels*), d'instructions et d'opérandes. Étant donnée une architecture $\mathtt{A}$: `arch`, ce code est interprété dans le langage intermédiaire $\mathtt{C}^{\iota} = [\![\mathtt{C}^{\alpha}]\!]_{\mathtt{A}} : \mathtt{asm}^{\iota}$ présenté en figure 4.2.

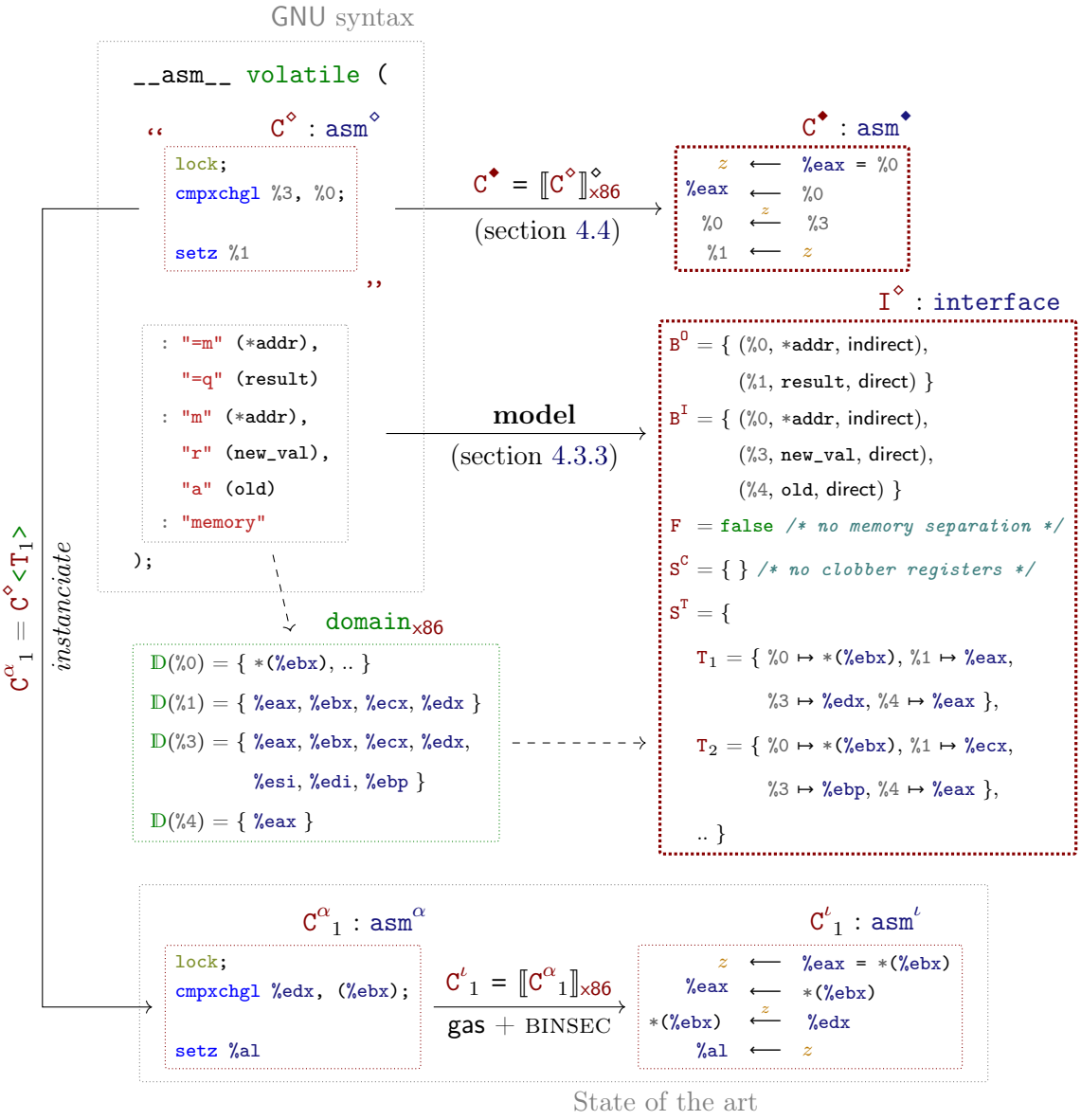


FIGURE 4.1 – Vue d'ensemble des définitions de types

$$
\begin{array}{rcl}
\texttt{asm}^{\iota} & := & \mathsf{begin} : \mu\texttt{label}\ ;\ \mathsf{body} : \mu\texttt{label} \rightarrow \mu\texttt{instr}\ ;\ \mathsf{end} : \mu\texttt{label} \\
\mu\texttt{instr} & := & \mu\texttt{oper}\ \mathsf{goto}\ \mu\texttt{label} \\
\mu\texttt{oper} & := & \texttt{lvalue} \leftarrow \texttt{expr}\ ;\ |\ \mathsf{if}\ \texttt{expr}\ \mathsf{then\ goto}\ \mu\texttt{label}\ ;\ | \\
\texttt{lvalue} & := & \texttt{variable}\ |\ \texttt{@[}\ \texttt{expr}\ \texttt{]} \\
\texttt{expr} & := & \texttt{bitvector}\ |\ \texttt{lvalue}\ |\ \texttt{unop expr}\ |\ \texttt{binop expr expr} \\
\texttt{unop} & := & \neg\ |\ -\ |\ \mathsf{zext}_n\ |\ \mathsf{sext}_n\ |\ \mathsf{extract}_{j..i} \\
\texttt{binop} & := & \texttt{arith}\ |\ \texttt{bitwise}\ |\ \texttt{cmp}\ |\ \mathsf{concat} \\
\texttt{arith} & := & +\ |\ -\ |\ \times\ |\ /_u\ |\ \mathsf{mod}_u\ |\ /_s\ |\ \mathsf{mod}_s \\
\texttt{bitwise} & := & \wedge\ |\ \vee\ |\ \oplus\ |\ \mathsf{lsl}\ |\ \mathsf{lsr}\ |\ \mathsf{asr} \\
\texttt{cmp} & := & =\ |\ <_u\ |\ <_s
\end{array}
$$

FIGURE 4.2 – Représentation intermédiaire de l'assembleur $\texttt{asm}^{\iota}$

Un morceau d'assembleur $\texttt{C}^{\iota}$ est un ensemble d'instructions $\mu\texttt{instr}$ identifiées par des étiquettes $\mu\texttt{label}$. Deux étiquettes particulières marquent le début et la fin du bloc. Chaque instruction peut réaliser une opération élémentaire (affectation ou saut conditionnel) avant de sauter à la prochaine étiquette. Seules les instructions ont un effet sur l'état du système, l'évaluation des expressions est sans effet de bord. Les éléments `lvalue` porteurs d'une valeur sont :

1. des variables pouvant représenter des registres publics (par exemple `%eax` ou `%ebx` pour le x86) ou des registres internes qui servent de variables temporaires ;
2. des cellules mémoire d'un octet, adressées par une expression standard.

Les variables servent également à représenter les étiquettes (aussi appelées symboles) de l'assembleur $\texttt{C}^{\alpha}$. Dans ce cas, elles ne seront, en pratique, jamais le membre gauche d'une affectation.

Les opérateurs de $\texttt{asm}^{\iota}$ sont standards et communs avec le `C`, à l'exception des opérateurs suivants empruntés au binaire (DBA [BHL+11]) :

— l'opérateur d'extension non signée, zext_n qui ajoute à son argument autant de bits mis à zéro pour que la taille du résultat soit de n bits ;

— l'opérateur d'extension signée, sext_n qui ajoute à son argument autant de bits égaux à son bit de poids fort pour que la taille du résultat soit de n bits ;

— l'opérateur d'extraction de bits, $\mathsf{extract}_{j..i}$ qui sélectionne les bits allant de l'indice i à l'indice j. L'indice j doit être plus petit que la taille de l'argument et plus grand ou égal que l'indice i. Le résultat obtenu est de taille $j - i + 1$.

— l'opérateur de concaténation, concat dont la taille du résultat est la somme des tailles de ses arguments et dont les bits de poids fort sont égaux à ceux de son premier argument et dont les bits de poids faible sont égaux à ceux de son deuxième argument.

Notation étendue. Par soucis de lisibilité, l'extraction de bits sera écrite avec des indices entre accolades : $\texttt{a}_{\{j..i\}} = \mathsf{extract}_{j..i}\ \texttt{a}$, simplifiée lorsque les indices de début

et de fin sont identiques en $\texttt{a}_{\{i\}} = \texttt{a}_{\{i..i\}}$. De plus, les opérateurs binaires `binop` seront écrits dans une notation infixe. L'opérateur de concaténation sera noté "::" en position infixe : `a :: b` = concat `a b`.

L'opérateur de choix ternaire `a ? b : c`, où `a` est un booléen (`bitvector` de taille 1) et `b` et `c` des `bitvectors` de même taille, est utilisé à la place de $((\mathsf{sext}_{\mathsf{sizeof(b)}}\ \texttt{a}) \wedge \texttt{b}) \vee ((\mathsf{sext}_{\mathsf{sizeof(c)}}\ (\neg\ \texttt{a})) \wedge \texttt{c})$.

Enfin, l'accès à des cellules mémoire contiguës peut être simplifié avec la notation petit-boutiste (*little endian*) $\texttt{@[a]}_{\overleftarrow{n}}$ ou gros-boutiste (*big endian*) $\texttt{@[a]}_{\overrightarrow{n}}$:

— $\texttt{@[a]}_{\overleftarrow{n}}$:= `@[a + n − 1] :: .. :: @[a + 1] :: @[a]` ;

— $\texttt{@[a]}_{\overrightarrow{n}}$:= `@[a] :: @[a + 1] :: .. :: @[a + n − 1]`.

La notation (abusive) $\texttt{@[a]}_{*}$ représente un accès mémoire contiguë de taille arbitraire défini de sorte que le reste de l'expression soit cohérent et que la convention générale de l'architecture soit respectée. Ainsi, `%eax` ← $\texttt{@[a]}_{*}$ signifie `%eax` ← $\texttt{@[a]}_{\overleftarrow{4}}$ pour le x86.

Notons également que les opérateurs sext_n et concat sont définissables à partir des autres opérateurs – voici un exemple :

— $\texttt{a}\ \mathsf{concat}\ \texttt{b} = \texttt{a} :: \texttt{b} = ((\mathsf{zext}_{\mathsf{sizeof(b)}}\ \texttt{a})\ \mathsf{shl}\ \mathsf{sizeof}(\texttt{b})) \vee (\mathsf{zext}_{\mathsf{sizeof(a)}}\ \texttt{b})$;

— $\mathsf{sext}_n\ \texttt{a} = (-(\mathsf{zext}_{n-\mathsf{sizeof(a)}}\ \texttt{a}_{\{\mathsf{sizeof(a)}-1\}})) :: \texttt{a}$.

Limitations. La représentation intermédiaire $\texttt{asm}^{\iota}$ impose au morceau d'assembleur $\texttt{C}^{\alpha}$ les contraintes suivantes :

1. le code assembleur $\texttt{C}^{\alpha}$ ne doit contenir que des instructions assembleur valides pour l'architecture `A`, déterministes et décomposables en une série de micro instructions $\mu\texttt{instr}$;
2. le code assembleur doit se comporter comme un bloc de base : il n'a qu'un seul point d'entrée et qu'un seul point de sortie et ne doit pas faire appel à du code extérieur.

Nous verrons en section 4.4.4 que ces restrictions ne sont, *empiriquement*, que peu limitantes sur notre corpus d'étude (section 3.2.3).

Le langage $\texttt{asm}^{\iota}$ ne supporte pas les sauts indirects permis par la syntaxe assembleur (e.g. `jump *eax`). Toutefois, les usages légitimes d'un saut indirect se résument généralement à deux cas. Le premier est un appel ou retour de fonction dynamique, ce qui sort du cadre de cette formalisation (c.f. point 2). Le second est la réalisation efficace d'un branchement multi-conditionnel (table de sauts). L'ensemble des cibles de saut est alors fini et il est possible d'exprimer ce saut dynamique dans le langage intermédiaire comme une succession de branchements conditionnels (e.g. if `%eax` = `s` then goto $\texttt{l}_{\texttt{s}}$).

4.2.2 Sémantique opérationnelle de l'assembleur $\texttt{asm}^{\iota}$

Nous mélangeons la sémantique bas niveau des *bitvecteurs* avec un modèle mémoire inspirée du C afin d'obtenir une sémantique hybride, capable de faire le lien

entre binaire et C. La mémoire est ainsi organisée en blocs dont l'adresse de base est traitée symboliquement et qui suit les mêmes restrictions que les pointeurs en C. Cependant, le code assembleur pouvant localement casser les invariants de ce modèle mémoire haut niveau, nous utilisons des techniques de normalisation inspirées des travaux de Besson et al. [BBW14] et Djoudi et al. [DB15] avec l'objectif d'éliminer les expressions qui sont temporairement indéfinies. Le principe général est de conserver la représentation symbolique des calculs ayant menés à une opération indéfinie localement, jusqu'à ce que ces calculs puissent être normalisés en expressions bien définies, ou déclenchent une erreur (accès mémoire, branchement).

L'assembleur agit sur un état mémoire $\mathtt{S} \triangleq (\mathtt{R}, \mathtt{M}) : \mathtt{state}$, composé d'un ensemble fini de registres $\mathtt{R} : \mathtt{variable} \mapsto \mathtt{value}$ et d'une mémoire composée de cellules adressées à l'octet $\mathtt{M} : \mathtt{addr} \mapsto \mathtt{value}$. Une valeur concrète $\mathtt{value} := \mathtt{bitvector} \mid \mathtt{addr} \mid \mathsf{undef}(\mathtt{expr}^{\#})$, peut être une chaîne de bits arbitraire, une adresse ou une expression indéfinie symbolique.

— Les opérations entre bitvectors sont standards et entièrement définies, à l'exception des divisions udiv et sdiv par 0 (et par extension, les calculs de reste urem et srem) ;

— Une adresse $\mathtt{addr} \triangleq \mathtt{base} \times \mathtt{bitvector}$ se compose d'une base $\mathtt{b} : \mathtt{base}$ et d'un décalage (*offset*) au sein de cette base. Une base $\mathtt{b} : \mathtt{base}$ est un identifiant unique associé à un espace mémoire dont la taille en octets est notée $\mathsf{sizeof}(\mathtt{b})$. À l'instar des pointeurs en C, seules l'addition (commutative) + et la soustraction − d'une base base par une constante bitvector produisent une adresse valide. La soustraction − de deux bases base identiques retourne la constante zéro ;

— Les autres opérations sont indéfinies et produisent des expressions symboliques $\mathsf{undef}(\mathtt{expr}^{\#})$, où $\mathtt{expr}^{\#} := \mathtt{bitvector} \mid \mathtt{addr} \mid \mathtt{unop}\ \mathtt{expr}^{\#} \mid \mathtt{binop}\ \mathtt{expr}^{\#}\ \mathtt{expr}^{\#}$, dont le rôle est de retarder l'évaluation au cas où de futures opérations la rendraient définie.

La fonction $\mathtt{eval}^{\#} : \mathtt{expr}^{\#} \mapsto \mathtt{expr}$ permet de forcer l'évaluation d'une expression symbolique et procède en deux temps :

1. elle normalise d'abord l'expression à l'aide de règles de réécriture telles que la commutativité ou l'associativité des opérateurs (la liste des règles de réécriture utilisées en pratique est donnée en annexe B) ;
2. elle évalue ensuite la nouvelle expression si cette dernière est entièrement définie ou, à défaut, renvoie une nouvelle valeur $\mathsf{undef}(\mathtt{expr}^{\#})$.

Les valeurs non définies sont ainsi propagées symboliquement jusqu'à provoquer une erreur si elles sont utilisées pour accéder à la mémoire ou calculer la cible d'un saut conditionnel, ou être réévaluées après avoir été normalisées. Ce dernier cas permet par exemple à l'expression suivante d'être correctement évaluée malgré le fait que l'addition de deux bases différentes ne soit pas définie :

$$\begin{aligned}\forall \mathtt{b}_1 : \mathtt{base}, \mathtt{b}_2 : \mathtt{base};\ & ((\mathtt{b}_1, 0) + (\mathtt{b}_2, 0)) - (\mathtt{b}_1, -4) \\ & \hookrightarrow ((\mathtt{b}_1, 0) - (\mathtt{b}_1, -4)) + (\mathtt{b}_2, 0) \\ & \hookrightarrow (\mathtt{b}_2, 4)\end{aligned}$$

La fonction `eval` : `state` × `expr` ↦ `value` permet d'évaluer une expression `e` dans un état mémoire `S` donné. Nous donnons la définition de la fonction `eval` en figure 4.3, en utilisant la notation $\overrightarrow{\texttt{eval}}$ pour forcer l'évaluation des expressions indéfinies, le cas échéant.

La sémantique opérationnelle (à grands pas) est donnée par la fonction `exec` : `state` × $\texttt{asm}^{\iota}$ ↦ `state`. Celle-ci repose sur `eval` pour évaluer les expressions et transformer l'état mémoire `M` en un nouvel état $\texttt{M}'$. Nous donnons la définition de la fonction `exec` en figure 4.4.

4.3 Spécificités de l'assembleur embarqué

La sémantique opérationnelle va nous servir de base pour pouvoir analyser les spécificités de l'assembleur embarqué, à savoir la substitution des jetons dans le patron et l'interface déclarative avec le C. La sémantique opérationnelle de l'assembleur embarqué est donnée dans le cadre du respect de la conformité à l'interface (voir chapitre 5).

4.3.1 Formalisation des patrons $\texttt{C}^{\diamond}$ et des jetons `t`

Un morceau d'assembleur embarqué n'utilise pas directement de l'assembleur standard $\texttt{C}^{\alpha}$: $\texttt{asm}^{\alpha}$. À la place, il utilise un patron (*template*) $\texttt{C}^{\diamond}$: $\texttt{asm}^{\diamond}$ dans lequel certains opérandes assembleur sont remplacés par des jetons (*token*) `t` : `token`. Un jeton est un identifiant (e.g. `%0`, `%1`, etc.) auquel sera substitué un opérande classique. Étant donné une architecture `A` : `arch`, un patron est interprété dans le langage intermédiaire $\texttt{C}^{\blacklozenge} = [\![\texttt{C}^{\diamond}]\!]^{\diamond}_{\texttt{A}}$: $\texttt{asm}^{\blacklozenge}$. Le langage intermédiaire $\texttt{asm}^{\blacklozenge}$ donné dans la figure 4.5 est en tout point similaire au langage $\texttt{asm}^{\iota}$ donné en figure 4.2 à l'exception de l'ajout du type `token` en tant qu'identifiant (règle $\texttt{lvalue}^{\blacklozenge}$).

Étant donné une affectation de jetons `T` : `token` ↦ `expr`, un patron $\texttt{C}^{\blacklozenge}$ peut être converti en assembleur classique $\texttt{C}^{\iota}$ à l'aide de la substitution syntaxique standard `<>`, dénotée $\texttt{C}^{\blacklozenge}\texttt{<T>}$: $\texttt{asm}^{\iota}$. Il est ainsi possible de produire un état $\texttt{S}'$ à partir d'un état `S`, d'un patron $\texttt{C}^{\blacklozenge}$ et d'une affectation de jetons `T` avec $\texttt{S}' = \texttt{exec}(\texttt{S}, \texttt{C}^{\blacklozenge}\texttt{<T>})$.

Nous définissons la valeur d'un jeton `t` à travers l'affectation `T` dans un état `S` par $\texttt{eval}(\texttt{S}, \texttt{T}(\texttt{t}))$.

4.3.2 Modélisation de l'interface formelle $\texttt{I}^{\diamond}$

L'interface réalise le lien entre les langages C et $\texttt{asm}^{\blacklozenge}$. Ne sont considérées ici que les expressions C sans effet de bord. L'interface fait référence à des expressions C

$$\overrightarrow{\texttt{eval}}(\texttt{S}, \texttt{e}) = \begin{cases} \texttt{x} & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \texttt{x} : \texttt{bitvector} \\ \texttt{a} & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \texttt{a} : \texttt{addr} \\ \texttt{eval}^{\#}(\texttt{u}) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \mathsf{undef}(\texttt{u}) \end{cases}$$

$$\texttt{eval}(\_, \texttt{x} : \texttt{bitvector}) = \texttt{x}$$

$$\texttt{eval}((\texttt{R}, \_), \texttt{v} : \texttt{variable}) = \texttt{R}(\texttt{v})$$

$$\texttt{eval}(\texttt{S} \triangleq (\_, \texttt{M}), \texttt{@[e : expr]}) = \begin{cases} \texttt{M}(\texttt{a}) & \text{when } \overrightarrow{\texttt{eval}}(\texttt{S}, \texttt{e}) = \texttt{a} \triangleq (\texttt{b}, \texttt{x}) \text{ and } \texttt{x} <_u \mathsf{sizeof}(\texttt{b}) \\ \mathsf{error} & \text{otherwise} \end{cases}$$

$$\texttt{eval}(\texttt{S}, (\texttt{o} : \texttt{unop})\ (\texttt{e} : \texttt{expr})) = \begin{cases} \texttt{o x} & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \texttt{x} \\ \mathsf{undef}(\texttt{o a}) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \texttt{a} \triangleq (\texttt{b}, \texttt{x}) \\ \mathsf{undef}(\texttt{o u}) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}) = \mathsf{undef}(\texttt{u}) \end{cases}$$

$$\texttt{eval}(\texttt{S}, (\texttt{e}_1 : \texttt{expr})\ (\texttt{o} : \texttt{binop})\ (\texttt{e}_2 : \texttt{expr})) = \begin{cases} \mathsf{error} & \text{when } \texttt{o} \in \{/_u, \mathsf{mod}_u, /_s, \mathsf{mod}_s\} \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = 0 \\ \texttt{x}_1\ \texttt{o}\ \texttt{x}_2 & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{x}_1 \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{x}_2 \\ (\texttt{b}, \texttt{x}_1\ \texttt{o}\ \texttt{x}_2) & \text{when } \texttt{o} \in \{+, -\} \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{a} \triangleq (\texttt{b}, \texttt{x}_1) \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{x}_2 \\ (\texttt{b}, \texttt{x}_1 + \texttt{x}_2) & \text{when } \texttt{o} = + \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{x}_1 \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{a} \triangleq (\texttt{b}, \texttt{x}_2) \\ \texttt{x}_1\ \texttt{o}\ \texttt{x}_2 & \text{when } \texttt{o} \in \{-, =, <_u, <_s\} \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{a}_1 \triangleq (\texttt{b}, \texttt{x}_1) \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{a}_2 \triangleq (\texttt{b}, \texttt{x}_2) \\ \mathsf{undef}(\texttt{u}_1\ \texttt{o}\ \texttt{u}_2) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \mathsf{undef}(\texttt{u}_1) \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \mathsf{undef}(\texttt{u}_2) \\ \mathsf{undef}(\texttt{u}\ \texttt{o}\ \texttt{v}) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \mathsf{undef}(\texttt{u}) \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{v} \\ \mathsf{undef}(\texttt{v}\ \texttt{o}\ \texttt{u}) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{v} \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \mathsf{undef}(\texttt{u}) \\ \mathsf{undef}(\texttt{v}_1\ \texttt{o}\ \texttt{v}_2) & \text{when } \texttt{eval}(\texttt{S}, \texttt{e}_1) = \texttt{v}_1 \\ & \text{and } \texttt{eval}(\texttt{S}, \texttt{e}_2) = \texttt{v}_2 \end{cases}$$

FIGURE 4.3 – Fonction $\texttt{eval}(\texttt{S}, \texttt{e})$

$$
\begin{aligned}
\texttt{exec}(\texttt{S},\, \texttt{C}^{\iota}) &= \texttt{step}(\texttt{S},\, \texttt{C}^{\iota},\, \texttt{C}^{\iota}.\mathsf{begin}) \\
\texttt{step}(\texttt{S},\, \texttt{C}^{\iota},\, \texttt{l}) &= \begin{cases} \texttt{S} \quad \text{when } \texttt{l} = \texttt{C}^{\iota}.\mathsf{end} \\ \mu\texttt{exec}(\texttt{S}, \texttt{C}^{\iota}, \texttt{C}^{\iota}.\mathsf{body}(\texttt{l})) \quad \text{otherwise} \end{cases} \\
\mu\texttt{exec}(\texttt{S} \triangleq (\texttt{R},\, \texttt{M}),\, \texttt{C}^{\iota},\, \texttt{v} \leftarrow \texttt{e}\,;\, \text{goto } \texttt{l}) &= \texttt{step}((\texttt{R}(\texttt{v} := \texttt{eval}(\texttt{S},\, \texttt{e})),\, \texttt{M}),\, \texttt{C}^{\iota},\, \texttt{l}) \\
\mu\texttt{exec}(\texttt{S} \triangleq (\texttt{R},\, \texttt{M}),\, \texttt{C}^{\iota},\, \texttt{@[}\texttt{e}_1\texttt{]} \leftarrow \texttt{e}_2\,;\, \text{goto } \texttt{l}) &= \begin{cases} \texttt{step}((\texttt{R}, \texttt{M}(\texttt{a} := \texttt{eval}(\texttt{S}, \texttt{e}_2))), \texttt{C}^{\iota}, \texttt{l}) \\ \quad \text{when } \overrightarrow{\texttt{eval}}(\texttt{S},\, \texttt{e}_1) = \texttt{a} \triangleq (\texttt{b},\, \texttt{x}) \text{ and } \texttt{x} <_u \mathsf{sizeof}(\texttt{b}) \\ \mathsf{error} \quad \text{otherwise} \end{cases} \\
\mu\texttt{exec}(\texttt{S},\, \texttt{C}^{\iota},\, \text{if } \texttt{e} \text{ then goto } \texttt{l}_1\,;\, \text{goto } \texttt{l}_2) &= \begin{cases} \texttt{step}(\texttt{S}, \texttt{C}^{\iota}, \texttt{l}_1) \quad \text{when } \overrightarrow{\texttt{eval}}(\texttt{S},\, \texttt{e}) = \texttt{0b1} \\ \texttt{step}(\texttt{S}, \texttt{C}^{\iota}, \texttt{l}_2) \quad \text{when } \overrightarrow{\texttt{eval}}(\texttt{S},\, \texttt{e}) = \texttt{0b0} \\ \mathsf{error} \quad \text{otherwise} \end{cases} \\
\mu\texttt{exec}(\texttt{S},\, \texttt{C}^{\iota},\, \text{goto } \texttt{l}) &= \texttt{step}(\texttt{S},\, \texttt{C}^{\iota},\, \texttt{l})
\end{aligned}
$$

FIGURE 4.4 – Fonction $\texttt{exec}(\texttt{S},\, \texttt{C}^{\iota})$

$$
\begin{aligned}
\texttt{asm}^{\blacklozenge} &:= \mathsf{begin} : \mu\texttt{label}\,;\, \mathsf{body} : \mu\texttt{label} \rightarrow \mu\texttt{instr}^{\blacklozenge}\,;\, \mathsf{end} : \mu\texttt{label} \\
\mu\texttt{instr}^{\blacklozenge} &:= \mu\texttt{oper}^{\blacklozenge} \text{ goto } \mu\texttt{label} \\
\mu\texttt{oper}^{\blacklozenge} &:= \texttt{lvalue}^{\blacklozenge} \leftarrow \texttt{expr}^{\blacklozenge}\,;\, \mid \text{if } \texttt{expr}^{\blacklozenge} \text{ then goto } \mu\texttt{label}\,;\, \mid \\
\texttt{lvalue}^{\blacklozenge} &:= \texttt{variable} \mid \texttt{@[expr}^{\blacklozenge} \texttt{]} \mid \texttt{token} \\
\texttt{expr}^{\blacklozenge} &:= \texttt{bitvector} \mid \texttt{lvalue}^{\blacklozenge} \mid \texttt{unop expr}^{\blacklozenge} \mid \texttt{binop expr}^{\blacklozenge}\ \texttt{expr}^{\blacklozenge}
\end{aligned}
$$

FIGURE 4.5 – Représentation intermédiaire des patrons

`expr`$^{\mathsf{c}}$ ainsi que sa sous-catégorie, les éléments `lvalue`$^{\mathsf{c}}$, auxquels on peut affecter une valeur. L'adresse d'un élément `lvalue`$^{\mathsf{c}}$ est donnée par l'opérateur unaire « & », noté `&v` pour `v` : `lvalue`$^{\mathsf{c}}$.

Une interface est modélisée par un tuple $\mathtt{I}^{\diamond} \triangleq (\mathtt{B}^{0}, \mathtt{B}^{\mathtt{I}}, \mathtt{S}^{\mathtt{T}}, \mathtt{S}^{\mathtt{C}}, \mathtt{F})$: `interface`.

Les deux liens (*binding*), $\mathtt{B}^{0}$: (`token` × `lvalue`$^{\mathsf{c}}$ × `kind`) `set` pour les sorties et $\mathtt{B}^{\mathtt{I}}$: (`token` × `expr`$^{\mathsf{c}}$ × `kind`) `set` pour les entrées, lient les jetons et les expressions C. Un lien peut être de deux sortes `kind` := direct | indirect. Par analogie avec le C, une lien direct se comporte comme un argument de fonction, il s'agit d'une copie, alors qu'un lien indirect se comporte comme un pointeur, il s'agit d'un accès à un emplacement mémoire. Informellement, un lien (`t`, `e`, direct) de $\mathtt{B}^{\mathtt{I}}$ signifie que le jeton `t` est initialisé par la valeur de `e` à l'entrée du bloc, un lien (`t`, `v`, direct) de $\mathtt{B}^{0}$ signifie que la variable `v` recevra la valeur de `t` à la sortie du bloc. Un lien (`t`, `v`, indirect), autant dans $\mathtt{B}^{\mathtt{I}}$ que dans $\mathtt{B}^{0}$, aura pour effet de lier par référence le jeton `t` et la variable `v` : `&t` = `&v`.

L'indicateur de séparation mémoire `F` : `bool` indique le type de synchronisation de la mémoire. Lorsqu'il est faux (`false`), l'ensemble de la mémoire de l'environnement C est accessible et modifiable par l'assembleur. À l'inverse, lorsqu'il est vrai (`true`), l'assembleur n'a aucun effet de bord autre que l'affectation des expressions liées dans $\mathtt{B}^{0}$.

L'ensemble de registres $\mathtt{S}^{\mathtt{C}}$: `variable set` (*clobber*) est un ensemble de registres libres d'être utilisés dans le morceau d'assembleur sans produire d'effet de bord. Ils ne sont pas initialisés à l'entrée du bloc et leur valeur n'est pas significative à la sortie.

L'ensemble d'affectation de jetons $\mathtt{S}^{\mathtt{T}}$: (`token` ↦ `expr`) `set` contient l'ensemble des affectations de jetons `T` autorisées pour la substitution dans le patron. Un ensemble vide n'autorise aucune substitution et résulte en une erreur de compilation.

4.3.3 De la syntaxe concrète GNU à notre formalisme

Le modèle formel $\mathtt{I}^{\diamond}$ est dérivable à partir de la syntaxe concrète d'un bloc assembleur embarqué (figure 4.1, flèche « *model* »).

$\mathtt{S}^{\mathtt{C}}$ et `F` sont dérivés directement à partir de la liste des *clobbers* : les registres présents dans cette liste sont ajoutés à $\mathtt{S}^{\mathtt{C}}$; si cette liste contient le mot clé `"memory"`, l'indicateur `F` est mis à `false`, sinon, il est par défaut à `true`.

Par défaut, le compilateur suppose que le bloc est sans effet de bord autre que ses sorties. Il s'agit du comportement opposé de celui des attributs de fonctions (e.g. `__attribute__((const))` ou `__attribute__((pure))`) pour lesquelles le compilateur fait des hypothèses conservatrices par défaut. De par ce choix, ce qui est absent de l'interface a autant de valeur informative que ce qui est présent. Cette syntaxe est orientée pour l'« optimisation » plus que pour la « sûreté ».

Les jetons sont définis (numérotés) en suivant l'ordre des éléments de l'interface concrète en commençant par les sorties puis en poursuivant par les entrées. Le terme est composé à partir du jeton nouvellement défini et de l'expression C entre

parenthèses. Le type de terme dépend du type de l'expression C. Si le type ne permet pas la copie (type incomplet tel que `int (*)[]`) ou possédant l'attribut `volatile`), le terme est indirect et la contrainte ne doit autoriser qu'un placement en mémoire ; sinon, il est direct par défaut.

Documentation GNU. La distinction entre un terme direct et indirect n'est pas mentionnée dans la documentation. La syntaxe n'est d'ailleurs pas à même de distinguer ces cas. Les compilateurs GCC et Clang traitent pourtant les termes différemment en fonction de la nature des opérandes autorisés par les contraintes. Ainsi, lorsque la contrainte n'autorise qu'un placement en mémoire, le terme est considéré indirect et il est nécessaire que l'expression possède une adresse (qui échappe par conséquent au compilateur). Dans la majorité des cas, faire ou non cette distinction ne change pas le comportement du programme. Dans certain cas, le fait que l'adresse échappe au compilateur de façon implicite peut produire du code moins performant. Il existe toutefois des cas pathologiques où le comportement du programme diffère selon l'interprétation. C'est le cas lorsqu'un tableau est donné en entrée d'un bloc assembleur avec la contrainte `"m"`. GCC et Clang prennent implicitement l'adresse du tableau (soit lui-même, en C) et l'opérande assembleur sélectionné pointe sur la même zone mémoire. ICC en revanche considère l'entrée comme direct : l'opérande sélectionné pointe sur une cellule mémoire locale où est stockée une copie de la valeur du tableau (à savoir, son adresse).
Dans la suite du document, l'interprétation de l'interface concrète suivra l'implémentation de GCC et Clang en considérant les contraintes de type mémoire exclusif (e.g. `"m"`) comme indirect.

L'ajout du terme à l'interface suit les règles suivantes :

1. Si la contrainte commence par le signe '`+`', le terme est ajouté à la fois à $\texttt{B}^{\texttt{O}}$ et $\texttt{B}^{\texttt{I}}$ (entrée/sortie) ;
2. Si la contrainte commence par le signe '`=`', le terme est ajouté à $\texttt{B}^{\texttt{O}}$ (sortie) ;
3. En l'absence des signes '`+`' et '`=`', le terme est ajouté à $\texttt{B}^{\texttt{I}}$ (entrée) ;
4. Si la contrainte d'une entrée ne lui permet pas d'être disjointe d'une sortie (par exemple `"=a"` avec `"a"`, la contrainte d'égalité `"0"`, etc), le jeton de la sortie est utilisé en lieu et place du jeton de l'entrée (*unified*).

Calculer l'ensemble d'affectation de jetons $\texttt{S}^{\texttt{T}}$ nécessite de résoudre le système de contraintes [2] posé par les entrées, les sorties et les *clobbers* de la syntaxe concrète. Le système de contraintes est construit comme une disjonction d'alternatives (séparées par des '`,`' dans la syntaxe). Une alternative est une conjonction de contraintes sur les jetons. Au sein d'une alternative, le modificateur '`&`' produit une conjonction d'équations de « non-inclusion » entre un jeton et un autre : $\texttt{t} \notin \texttt{t}'$ signifie que si `e` et $\texttt{e}'$ sont les opérandes associés respectivement à `t` et $\texttt{t}'$, alors `e` ne doit pas être une sous-expression de $\texttt{e}'$. Les caractères numériques ('0'-'9') produisent une

2. En pratique, il est préférable de calculer paresseusement l'ensemble $\texttt{S}^{\texttt{T}}$ en travaillant sur une sur- ou sous-approximation, ce que fait TInA (section 4.4.3).

disjonction d'égalités entre deux jetons ($\texttt{t} = \texttt{t}'$). Les autres caractères supportés par l'architecture (par exemple, ceux de la figure 3.5) produisent une disjonction d'égalités entre un jeton et un opérande assembleur ($\texttt{t} = \texttt{e}$). Chaque alternative est enrichie d'une conjonction d'équations de non-inclusion entre chaque registre *clobber* et chaque jeton, $\texttt{r} \notin \texttt{t}$.

Une façon de procéder est d'énumérer les domaines de chaque jeton pour chaque alternative (figure 4.1, boîte « `domain` »). Le domaine d'un jeton `t` pour une alternative `i` donnée est composé de l'ensemble des égalités `t = e` qui apparaissent dans `i`. Dans le cas particulier d'une égalité de jetons, le domaine inclut alors l'ensemble des opérandes supportés par l'architecture `A` (`top`). Ces ensembles sont combinés à l'aide du produit cartésien, produisant un ensemble de tuples associant un jeton à un opérande. Une opération de filtrage retire ensuite les tuples qui ne respectent pas les contraintes d'égalité ou de non-inclusion. Le résultat $\texttt{S}^{\texttt{T}}$ est l'union pour toutes les alternatives des tuples obtenus.

Si $\texttt{domain}_{\texttt{A}}$: `char` ↦ `expr set` est la fonction qui pour chaque caractère retourne l'ensemble d'opérandes associés pour l'architecture `A` (dont la figure 3.5 illustre le cas du x86), la table `constraints[token][int]` ↦ `char set` retourne les caractères présents dans la contrainte d'un jeton `t` pour une alternative `i` donnée, `n` : `int` le nombre d'alternatives et Φ : (`token` ↦ `expr`) ↦ `bool` la fonction qui vérifie la cohérence d'une affectation de jetons `T` par rapport aux contraintes d'inégalité et de non-inclusion, l'ensemble $\texttt{S}^{\texttt{T}}$ est défini par la formule suivante :

$$\texttt{S}^{\texttt{T}} = \bigcup_{\texttt{i}=0}^{\texttt{n}-1} \left\{ \texttt{T} \in \left(\bigtimes_{\texttt{t} \in \texttt{token}} \bigcup_{\texttt{l} \in \texttt{constraints[t][i]}} \texttt{domain}_{\texttt{A}}(\texttt{l}) \right) \middle| \Phi(\texttt{T}) \right\}$$

4.3.4 Sémantique opérationnelle de l'assembleur embarqué $\texttt{asm}^{\blacklozenge}$

L'objectif de cette section est de définir l'effet de l'exécution d'un bloc assembleur embarqué dans un environnement C *sans passer par une étape de compilation.*

Le langage C est générique et largement paramétrable (*implementation defined*). Cependant, à partir du moment où le code C incorpore de l'assembleur embarqué d'une architecture `A`, la taille de ses types scalaires (`char`, `short`, `int`, etc) et de ses pointeurs est fixée en conséquence pour permettre l'interopérabilité avec cet assembleur.

La définition de la sémantique opérationnelle repose sur deux hypothèses fortes :

1. la première hypothèse concerne l'interface du bloc assembleur :
   - l'interface doit être complète dans le sens où toutes les écritures et lectures sont correctement listées ;
   - l'allocation des jetons ne doit pas influencer le comportement des instructions présentes dans le patron $\texttt{X}^{\diamond}$.

   Ce prérequis est nommé « *conformité à l'interface* » et le chapitre 5 est dédié à sa définition précise et aux moyens de la vérifier. Grâce à cette conformité à

l'interface, il n'est plus nécessaire de distinguer un jeton d'une variable standard ;

2. la seconde hypothèse est le principe de « *transparence* » de l'assembleur. Il s'agit d'un concept similaire à la notion de fonction *pure* et stipule que le résultat de l'exécution ne dépend que de l'environnement C donné en entrée. Autrement dit, le code assembleur ne doit pas avoir d'état mutable caché.

Validité des hypothèses. La *transparence* est une propriété qui se vérifie empiriquement bien sur les exemples rencontrés lors de l'étude (voir section 3.2.3). De plus, il est possible dans la majorité des cas de transformer un bloc assembleur non transparent en un bloc transparent en externalisant son état privé qui serait alors donné à travers une nouvelle interface.

Ces deux hypothèses permettent de définir simplement la fonction $\texttt{exec}^{\diamond} : \texttt{scope} \times \texttt{interface} \times \texttt{asm}^{\blacklozenge} \mapsto \texttt{scope}$ qui relie l'état courant du programme C et l'état après l'exécution du bloc.

La fonction de transfert prend en entrée un environnement $\texttt{E} : \texttt{scope}$, une interface $\texttt{I}^{\diamond} : \texttt{interface}$ et un code assembleur $\texttt{C}^{\diamond} : \texttt{asm}^{\blacklozenge}$ et renvoie un nouvel environnement $\texttt{E}' : \texttt{scope}$.

Un environnement $\texttt{E} \triangleq (\texttt{V}, \texttt{M}) : \texttt{scope}$ est composé d'un ensemble de variables locales $\texttt{V} : \texttt{id} \mapsto \texttt{value}$ et d'un ensemble de blocs mémoire $\texttt{M} : \texttt{base} \times \texttt{bitvector} \mapsto \texttt{value}$. Les blocs mémoire incluent les variables globales [3] et la mémoire allouée dynamiquement.

La fonction $\texttt{exec}^{\diamond}$ se décompose en trois phases : l'initialisation de l'état mémoire assembleur, l'exécution de l'assembleur et la collecte des résultats. Elle dépend de trois nouvelles fonctions :

— $\texttt{alloc}^{\texttt{T}} : \texttt{interface} \mapsto (\texttt{token} \mapsto \texttt{expr})$;

— $\texttt{init} : \texttt{scope} \times \texttt{interface} \mapsto \texttt{state}$;

— $\texttt{collect} : \texttt{scope} \times \texttt{interface} \times \texttt{state} \mapsto \texttt{scope}$.

Ces fonctions seront définies plus loin. Nous définissons formellement $\texttt{exec}^{\diamond}$ comme :

$$\texttt{exec}^{\diamond}(\texttt{E}, \texttt{I}^{\diamond}, \texttt{C}^{\blacklozenge}) = \texttt{collect}(\texttt{E}, \texttt{I}^{\diamond}, \texttt{exec}(\texttt{init}(\texttt{E}, \texttt{I}^{\diamond}), \texttt{C}^{\blacklozenge}\texttt{<T>})) \quad \texttt{with T} = \texttt{alloc}^{\texttt{T}}(\texttt{I}^{\diamond})$$

L'évaluation de l'interface avec le C repose sur trois fonctions classique que nous utilisons sans les définir ici, à savoir :

— la fonction $\texttt{eval}^{\texttt{c}} : \texttt{scope} \times \texttt{expr}^{\texttt{c}} \mapsto \texttt{value}$ renvoie la valeur associée à une expression C ;

— la fonction $\texttt{addressof} : \texttt{scope} \times \texttt{lvalue}^{\texttt{c}} \mapsto \texttt{addr}$ renvoie l'adresse (&) d'un emplacement C ;

— la fonction $\texttt{assign}^{\texttt{c}} : \texttt{scope} \times \texttt{lvalue}^{\texttt{c}} \times \texttt{value} \mapsto \texttt{scope}$ met à jour l'environnement E avec la nouvelle valeur v de l'emplacement l. Ainsi $\texttt{eval}^{\texttt{c}}(\texttt{assign}^{\texttt{c}}(\texttt{E}, \texttt{l}, \texttt{v}), \texttt{l}) = \texttt{v}$.

3. Y compris les variables locales dont l'adresse a échappée au contexte C, par exemple, lorsque son adresse a été donnée en argument d'une fonction externe.

Grâce à l'hypothèse de conformité à l'interface, l'association avec les éléments bas niveau de l'interface $\mathtt{I}^{\diamond}$ peut être ignorée. L'interface se résume donc aux liens d'entrée $\mathtt{B}^{\mathtt{I}}$ et de sortie $\mathtt{B}^{\mathtt{O}}$.

Nous utilisons également sans la définir la fonction $\mathtt{varof} : \mathtt{token} \mapsto \mathtt{variable}$ qui renvoie un nouvel identifiant unique (variable locale) pour un jeton donné.

Nous pouvons maintenant définir les trois fonctions dont dépend $\mathtt{exec}^{\diamond}$:

Allocation : la fonction $\mathtt{alloc} : \mathtt{token} \times \mathtt{kind} \mapsto \mathtt{expr}$ renvoie l'expression à substituer dans le code assembleur $\mathtt{C}^{\blacklozenge}$. La fonction $\mathtt{alloc}^{\mathtt{T}} : \mathtt{interface} \mapsto (\mathtt{token} \mapsto \mathtt{expr})$ génère l'affectation de l'ensemble de jetons $\mathtt{T}$ à partir du produit de l'appel de la fonction $\mathtt{alloc}$ sur l'ensemble des jetons des termes de $\mathtt{B}^{\mathtt{I}}$ et $\mathtt{B}^{\mathtt{O}}$;

$$\mathtt{alloc}(\mathtt{t}, \mathsf{direct}) = \mathtt{varof}(\mathtt{t})$$

$$\mathtt{alloc}(\mathtt{t}, \mathsf{indirect}) = \mathtt{@}[\mathtt{varof}(\mathtt{t})]_*$$

$$\mathtt{alloc}^{\mathtt{T}}(\mathtt{I}^{\diamond} \triangleq (\mathtt{B}^{\mathtt{O}}, \mathtt{B}^{\mathtt{I}}, \_, \_, \_)) = \bigcup_{(\mathtt{t},\_,\mathtt{k}) \in \mathtt{B}^{\mathtt{I}} \cup \mathtt{B}^{\mathtt{O}}} (\mathtt{t} \mapsto \mathtt{alloc}(\mathtt{t}, \mathtt{k}))$$

Initialisation : La fonction $\mathtt{init} : \mathtt{scope} \times \mathtt{interface} \mapsto \mathtt{state}$ initialise un état mémoire assembleur $\mathtt{S}$ à partir des termes présents dans l'interface $\mathtt{I}^{\diamond}$ et de l'environnement $\mathtt{E} : \mathtt{scope}$. Elle initialise les entrées directes présentes dans $\mathtt{B}^{\mathtt{I}}$ mais également les adresses des entrées et sorties indirectes présentes dans $\mathtt{B}^{\mathtt{I}}$ et $\mathtt{B}^{\mathtt{O}}$;

$$\mathtt{init}(\mathtt{E} \triangleq (\_, \mathtt{M}), \mathtt{I}^{\diamond} \triangleq (\mathtt{B}^{\mathtt{O}}, \mathtt{B}^{\mathtt{I}}, \_\ ..)) = (\mathtt{init}^{\mathtt{O}}(\mathtt{E}, \mathtt{B}^{\mathtt{O}}, \mathtt{init}^{\mathtt{I}}(\mathtt{E}, \mathtt{B}^{\mathtt{I}}, \varnothing)), \mathtt{M})$$

$$\begin{aligned}
\mathtt{init}^{\mathtt{O}}(_, \varnothing, \mathtt{R}) &= \mathtt{R} \\
\mathtt{init}^{\mathtt{O}}(\mathtt{E}, \{\ (_, _, \mathsf{direct})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{init}^{\mathtt{O}}(\mathtt{E}, \mathtt{K}, \mathtt{R}) \\
\mathtt{init}^{\mathtt{O}}(\mathtt{E}, \{\ (\mathtt{t}, \mathtt{l}, \mathsf{indirect})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{init}^{\mathtt{O}}(\mathtt{E}, \mathtt{K}, \mathtt{R}(\mathtt{varof}(\mathtt{t}) := \mathtt{addressof}(\mathtt{E}, \mathtt{l}))
\end{aligned}$$

$$\begin{aligned}
\mathtt{init}^{\mathtt{I}}(_, \varnothing, \mathtt{R}) &= \mathtt{R} \\
\mathtt{init}^{\mathtt{I}}(\mathtt{E}, \{\ (\mathtt{t}, \mathtt{e}, \mathsf{direct})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{init}^{\mathtt{I}}(\mathtt{E}, \mathtt{K}, \mathtt{R}(\mathtt{varof}(\mathtt{t}) := \mathtt{eval}^{\mathtt{c}}(\mathtt{E}, \mathtt{e})) \\
\mathtt{init}^{\mathtt{I}}(\mathtt{E}, \{\ (\mathtt{t}, \mathtt{l}, \mathsf{indirect})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{init}^{\mathtt{I}}(\mathtt{E}, \mathtt{K}, \mathtt{R}(\mathtt{varof}(\mathtt{t}) := \mathtt{addressof}(\mathtt{E}, \mathtt{l}))
\end{aligned}$$

Collection : La fonction $\mathtt{collect} : \mathtt{scope} \times \mathtt{interface} \times \mathtt{state} \mapsto \mathtt{scope}$ collecte le résultat de l'exécution de l'assembleur $\mathtt{S}$ et met à jour les emplacements de sortie du blocs ($\mathtt{B}^{\mathtt{O}}$) dans l'environnement $\mathtt{E}$.

$$\mathtt{collect}(\mathtt{E} \triangleq (\mathtt{V}, \_), \mathtt{I}^{\diamond} \triangleq (\mathtt{B}^{\mathtt{O}}, \_\ ..), \mathtt{S} \triangleq (\mathtt{R}, \mathtt{M})) = \mathtt{set}((\mathtt{V}, \mathtt{M}), \mathtt{B}^{\mathtt{O}}, \mathtt{R})$$

$$\begin{aligned}
\mathtt{set}(\mathtt{E}, \varnothing, _) &= \mathtt{E} \\
\mathtt{set}(\mathtt{E}, \{\ (\mathtt{t}, \mathtt{l}, \mathsf{direct})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{set}(\mathtt{assign}^{\mathtt{c}}(\mathtt{E}, \mathtt{l}, \mathtt{R}(\mathtt{varof}(\mathtt{t}))), \mathtt{K}, \mathtt{R}) \\
\mathtt{set}(\mathtt{E}, \{\ (_, _, \mathsf{indirect})\ \} \cup \mathtt{K}, \mathtt{R}) &= \mathtt{set}(\mathtt{E}, \mathtt{K}, \mathtt{R})
\end{aligned}$$

4.4 TINA : obtenir la représentation intermédiaire

La première étape de l'analyse de code assembleur embarqué consiste à obtenir une représentation intermédiaire appropriée.

À l'exception de l'outil vx86, développé par Maus [MMS08] et qui travaille directement sur un sous-ensemble de l'assembleur x86 MASM, les outils développés ces

dernières décennies se concentrent sur la transformation de code binaire en représentation intermédiaire (*binary lifter*).

Cette section explique la méthode adoptée par notre prototype TInA pour extraire efficacement la représentation intermédiaire de l'assembleur embarqué à partir de la coopération d'une chaîne d'outils déjà existants, orchestrée par trois nouveaux modules.

4.4.1 Architecture générale

Nous voulons être en mesure de raisonner sur le bloc assembleur indépendamment de l'affectation de jetons choisie par le compilateur. Notre objectif est par conséquent d'obtenir une représentation intermédiaire qui intègre la notion de jeton : $\texttt{C}^{\blacklozenge}$ (figure 4.1, boîte « $\texttt{asm}^{\blacklozenge}$ »). Conserver cette information depuis la forme textuelle du patron $\texttt{C}^{\diamond}$ demanderait un effort considérable d'ingénierie ad-hoc pour associer à chaque instruction assembleur sa représentation dans le langage $\texttt{asm}^{\blacklozenge}$. Nous proposons à la place une méthode pour retrouver dans le langage $\texttt{asm}^{\iota}$ la position des jetons (section 4.4.2) et qui permet ainsi de réutiliser les plateformes d'analyse de code binaire tel que BINSEC.

La composante de TInA responsable de l'extraction de la représentation intermédiaire s'articule donc autour d'un logiciel assembleur générique (gas par exemple) pour instancier le patron $\texttt{C}^{\alpha}$ et d'une plateforme d'analyse de binaire (*binary lifter*) pour en extraire la représentation intermédiaire $\texttt{C}^{\iota}$. La figure 4.6 donne une vue d'ensemble du processus.

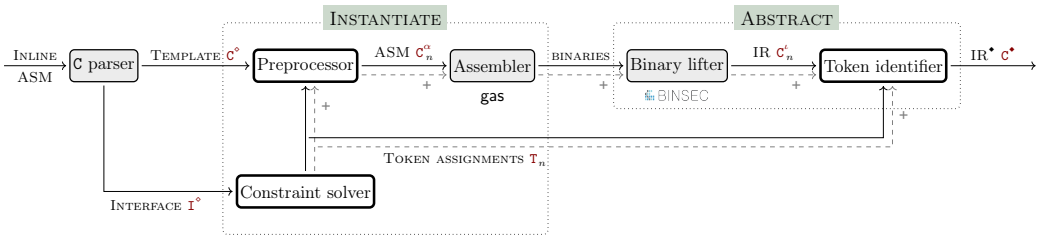


Figure 4.6 – Vue d'ensemble

Le morceau de code assembleur embarqué est décomposé à l'aide d'un analyseur syntaxique de C (*parser*) en un patron $\texttt{C}^{\alpha}$ (*template*) et son interface $\texttt{I}^{\diamond}$. Les contraintes de l'interface sont résolues par un **analyseur de contraintes** (*constraint solver*) dont le rôle est de fournir un petit nombre d'affectations de jetons valides (*token assignment*). Un **préprocesseur** se charge ensuite de substituer les jetons dans le patron par ces affectations. Le résultat de la substitution est donné au logiciel d'assemblage qui produit un code objet (*binary*). Ce code objet est ensuite désassemblé par la plateforme d'analyse de binaire (*binary lifter*) pour obtenir une représentation intermédiaire du code $\texttt{C}^{\iota}$. L'opération est répétée plusieurs fois ($+$) en faisant varier l'affectation de jetons, afin de générer plusieurs variants de la représentation intermédiaire qui seront en tout point similaire, à l'exception des opérandes assembleur situés à la position des jetons. Une **analyse différentielle** (*token iden-*

tifier) permet ensuite de détecter et d'unifier les positions de jetons qui diffèrent pour les remplacer par des variables spéciales.

Les sections 4.4.2 et 4.4.3 détaillent deux des trois nouveaux composants[4] de cette approche – l'identification de jetons et l'analyseur de contraintes. Le préprocesseur et l'identification de jetons sont génériques et indépendants de l'architecture cible. La conception permet ainsi de réduire le coût d'ingénierie au strict minimum, garantissant une portabilité maximale entre familles d'architectures et réduisant le risque d'erreurs de développement.

4.4.2 Identification de jetons

L'identification de jetons consiste à inverser l'opération de substitution des jetons par des opérandes. Elle prend en entrée une affectation de jetons et en inverse la relation.

Problème. Le compilateur peut choisir un même opérande pour plusieurs jetons : il peut par exemple partager le même registre pour plusieurs entrées qui ont la même valeur ou alors utiliser le même registre pour un jeton d'entrée et un jeton de sortie – l'affectation de registre $\texttt{T}_1$ de la figure 4.1 illustre ce cas où le registre `%eax` est utilisé à la fois pour `%1` et `%4`. Il ne faut pas non plus exclure le cas où un opérande était déjà présent « en dur » dans le code (*hard-coded*), auquel cas il ne doit pas être substitué par un jeton. Par conséquent, le résultat de l'inversion de l'affectation est une relation (*map*) d'opérandes vers un ensemble de jetons ou de registres. Pour une affectation de jetons donnée, nous appellerons *jeton ambigus* les jetons qui partagent le même antécédent (opérande assembleur).

Notre solution. Afin de lever l'ambiguïté de cette inversion, l'opération d'identification opère de façon incrémentale. La première itération permet de détecter les emplacements possibles des jetons au sein de la représentation intermédiaire et de les remplacer par des variables temporaires spéciales. Ensuite, pour chacun des jetons ambigus, une nouvelle affectation de jetons est demandée tel que ce jeton ne soit plus associé à ce même opérande. Si à l'emplacement d'un conflit, le code n'a pas changé, le jeton est retiré de l'ensemble d'ambiguïté et l'opération est réitérée jusqu'à ce que l'ensemble devienne un singleton. Si au contraire, le code a été modifié, le conflit est résolu et la variable temporaire est remplacée par ce jeton.

Dans le pire des cas, pour un ensemble de n jetons ambigus, le nombre d'itérations de l'algorithme est borné par $n + 1$, soit l'itération initiale, $n - 1$ itérations pour distinguer les jetons entre eux et une itération supplémentaire pour distinguer un jeton d'un registre écrit en dur. Si dès la première itération tous les jetons ont un antécédent unique, alors seules deux itérations sont nécessaires pour identifier avec certitude les jetons – en pratique TInA se place dans ce cas favorable sur l'ensemble des blocs assembleurs étudiés.

4. Le préprocesseur consiste simplement à substituer syntaxiquement des chaînes de caractères.

Les jetons « constants » (*immediate*) peuvent poser problème à cause des simplifications (*constant folding*) réalisées par l'outil d'assemblage (gas) ou la plateforme d'analyse de binaire. Dans ce cas, la valeur de la constante ne sera pas retrouvée comme le serait un registre. Il est toutefois possible de continuer à raisonner lorsque un conflit apparaît entre deux variants (valeur différente ou absence d'un côté ou de l'autre). Lors du choix de l'affectation de jetons, TInA choisie des valeurs de constantes de telle sorte à permettre la résolution de l'ambiguïté : les valeurs associées au jeton numéro i sont $i+1$ et $2 \times i + 1$. La soustraction des deux constantes en conflit retourne ainsi le numéro du jeton dont il est question.

4.4.3 Analyseur de contraintes

Le rôle de l'analyseur de contraintes est d'énumérer plusieurs affectations de jetons valides. Pour ce faire, il est nécessaire de connaître l'association entre les caractères présents dans les contraintes et les opérandes assembleur (le cas du x86 est donné en figure 3.5). Cette association est définie dans la documentation officielle de GNU. Certaines contraintes sont génériques et communes à toutes les architectures. Les contraintes dépendantes de l'architecture sont centralisées sur la page Machine Constraints [GCC20] de la documentation. Le support d'une architecture consiste à gérer, en moyenne, une vingtaine de contraintes différentes.

Comme expliqué en section 4.4.2, l'analyseur de contraintes n'a pas besoin, en pratique, d'être complet (c'est à dire, de savoir énumérer toutes les affectations valides). D'un part, le nombre d'affectations de jetons nécessaire à l'identification de jetons pour que le résultat ne soit pas ambigu est en effet borné par $n+1$, où n est le nombre de jetons – la syntaxe (section 3.2.2) imposant une limite d'au plus 30 jetons. De plus, si tous les jetons sont distincts les uns des autres, seules deux affectations de jetons sont nécessaires. Pour cette dernière raison, l'analyseur de contraintes à intérêt à favoriser une méthode d'allocation « agressive » où chaque jeton se voit assigner un opérande distinct.
Le problème est suffisamment petit (30 jetons maximum avec une moyenne à 4) pour utiliser une stratégie d'affectation gloutonne. L'analyseur de contraintes procède ainsi, jeton par jeton, en choisissant au hasard un opérande parmi l'ensemble des opérandes valides. En cas de détection d'une violation de contraintes, un retour en arrière est effectué. Afin de limiter le nombre de variants générés, l'analyseur de contraintes considère dans un premier temps que tous les jetons doivent être distincts. La section 4.4.4 confirme que cette stratégie agressive est très efficace en pratique puisqu'elle n'a jamais échouée sur l'architecture x86, dont le nombre de registres est pourtant très limité (7 registres généraux).

4.4.4 Évaluation expérimentale

L'extraction de la sémantique est implémentée dans notre prototype TInA, développé en OCaml (environ 2.5 kLOC pour l'extraction). Il utilise Frama-C [KKP+15]

pour analyser syntaxiquement le code C, l'assembleur GNU de base gas pour produire du code objet et la plateforme d'analyse de binaire BINSEC [DB15] pour le désassemblage.

La première version du prototype, dont les résultats de notre article ASE 2019 [RBB+19] sont issus, utilisait une approche différente pour l'extraction de la représentation intermédiaire. Elle se basait sur la chaîne complète du compilateur (GCC ou Clang) plutôt que d'utiliser directement l'assembleur (gas) et nécessitait de retrouver les informations sur l'affectation de jetons à l'aide des informations de « *debug* » DWARF. L'extraction dépendait ainsi de la qualité des informations DWARF produites par le compilateur, qui pouvaient varier en fonction des versions et du niveau d'optimisation. La nouvelle approche décrite ici ne souffre plus de ces inconvénients.

Appliqué au corpus d'évaluation x86, TInA est capable d'extraire la représentation intermédiaire de 2656 morceaux d'assembleur embarqué, soit environ **85**% des 3107 blocs rencontrés. Sur l'architecture ARM, TInA est capable d'extraire la représentation intermédiaire de 392 morceaux d'assembleur embarqué, soit la quasi-totalité (**99**%) des 394 blocs rencontrés. Le tableau 4.1 résume différentes statistiques sur les blocs assembleur.

Ces expérimentations montrent une grande efficacité dans l'extraction de la représentation intermédiaire par TInA. Toutefois, 373 blocs (environs 12% du total) utilisent des opérations système ou matériel qui sortent du cadre de TInA. De plus, 40 blocs (1%) réalisent des opérations en virgule flottante qui ne sont pas traitées par notre outil de désassemblage et 38 (1%) blocs utilisent des jeux d'instructions qui ne sont pas encore reconnus. Ces blocs sont peu nombreux, mais ils seraient un ajout intéressant aux expérimentations de TInA puisque qu'ils représentent 40% des 100 plus gros blocs assembleur du corpus.

4.5 Discussion

4.5.1 Représentativité du corpus d'exemple

Le code assembleur considéré est quantitativement et qualitativement représentatif de l'usage que font les développeurs de l'assembleur embarqué dans le code C. Il comprend tous les blocs assembleurs x86 trouvés dans le code source de la distribution Debian *Jessie* qui était la version stable au moment de commencer les travaux de thèse. Il est peu probable que cette base de code assembleur ait fondamentalement changée depuis. Les tests sur l'architecture ARM permettent de s'assurer que la technique est bien générique et s'adapte sans difficulté à différentes architectures. Les codes propriétaires sont évidemment exclus de ces expérimentations mais nous sommes confiants dans le fait que notre approche est suffisamment robuste et générique pour se comporter de la même façon.

TABLE 4.1 – Application de TInA sur l'assembleur embarqué de Debian

(a) Corpus x86

	TOTAL	ALSA	FFMPEG	GMP	libyuv
Blocs assembleur	3107	25	103	237	4
Supportés	2656 85%	25 100%	91 88%	237 100%	1 25%
Taille moyenne (Max)	3 (104)	69 (104)	12 (68)	1 (1)	40 (40)
# blocs de base moyen (Max)	1 (21)	12 (21)	2 (8)	1 (5)	3 (3)
Nombre de jetons (Max)	3 (12)	7 (8)	3 (6)	5 (8)	4 (4)
Allocation gloutonne	100%	100%	100%	100%	100%
Opérations système	373 12%	0 0%	4 4%	0 0%	3 75%
Taille moyenne (Max)	4 (151)	–	10 (21)	–	6 (12)
Virgules flottantes	40 1%	0 0%	5 5%	0 0%	0 0%
Taille moyenne (Max)	33 (506)	–	19 (38)	–	–
Autres limitations outil	38 1%	0 0%	3 3%	0 0%	0 0%
Taille moyenne (Max)	3 (31)	–	19 (31)	–	–

(b) Corpus ARM

	TOTAL	ALSA	FFMPEG	GMP	libyuv
Blocs assembleur	394	0	85	308	1
Supportés	392 99%	–	83 98%	308 100%	1 100%
Taille moyenne (Max)	1 (27)	–	1 (15)	1 (1)	27 (27)
# blocs de base moyen (Max)	2 (9)	–	1 (9)	2 (8)	4 (4)
Nombre de jetons (Max)	4 (7)	–	3 (4)	4 (7)	4 (4)
Allocation gloutonne	100%	–	100%	100%	100%
Opérations système	2 1%	–	2 2%	0 0%	0 0%
Taille moyenne (Max)	4 (6)	–	4 (6)	–	–

4.5.2 Limitations

Notre approche est principalement limitée par le support des jeux d'instructions de la plateforme BINSEC utilisée pour le désassemblage. En particulier, BINSEC manque de support sur les instructions de type système et dont le comportement apparaît comme non déterministe. Il manque également le support des instructions manipulant des nombres en virgule flottante (`float`).

L'ajout de ces instructions dans le décodeur pourrait être fait à différents niveaux :

— un support complet modélisant précisément tous les comportements de l'instruction et les différents registres matériels de l'architecture ;

— un support partiel précisant seulement une sur- ou sous-approximation, en fonction des besoins, des variables modifiées.

Toutefois, ces restrictions du décodeur ne sont que peu limitantes dans notre objectif de sûreté et de vérification de code C car ces deux types d'instructions ne sont que peu présents dans le code rencontré (15%).

4.6 Comparaison à l'état de l'art

Sémantique opérationnelle (formelle). Dans leurs travaux pionniers, Leroy et Blazy [LB08] proposent la première sémantique opérationnelle du C. Celle-ci suit les restrictions de la norme ANSI et ne peut donc pas représenter les opérations de bas niveau sur les pointeurs comme la soustraction de deux pointeurs aux bases différentes ou l'alignement à l'aide de masque bit à bit. Les travaux de Besson et al. [BBW17] prennent en compte quant à eux des éléments de bas niveau en autorisant la conversion des pointeurs vers une représentation sous forme d'entier dans certains cas. Se rapprochant du code assembleur, les travaux de Bardin et Herrmann [BH08], de Kinder et al. [KZV09] et de Balakrishnan et al. [BR10] ont posé les bases d'une sémantique opérationnelle très bas niveau du code binaire. En particulier, le modèle mémoire est assimilé à un grand tableau d'octets (modèle plat). Djoudi et Bardin [DB15] proposent de retrouver une structure haut niveau à l'aide de régions mémoire avec des règles de réécriture rappelant celles de Besson et al. [BBW14].

Notre approche se place à un niveau hybride. Les opérations sur les données suivent la sémantique bas niveau des bitvecteurs (pas de comportements indéfinis sur les signés) mais le modèle mémoire est directement hérité du C car l'assembleur embarqué manipule les données de l'environnement C sans en modifier la structure (pas d'allocations/désallocations). Notre gestion symbolique des bases de blocs mémoire est inspirée des travaux de Besson et al. [BBW14] et de Djoudi et al. [DB15] pour autoriser certains types d'opérations normalement indéfinis sur la représentation des pointeurs, tant que le résultat final est défini au niveau C.

L'assembleur embarqué en pratique. Maus [MMS08] traite la syntaxe Microsoft et est par conséquent limité au jeu d'instruction x86-32. Sa technique est

équivalente à virtualiser les codes assembleur à l'aide d'un interpréteur écrit en C : l'état de la machine est déclaré en variable globale et une fonction (*handler*) est associée à chacune des instructions assembleur. L'assembleur est transformé par une passe de prétraitement syntaxique qui remplace textuellement les instructions assembleur reconnues par l'appel à la fonction associée.

Fehnker et al. [FHRS08] utilisent un analyseur syntaxique dédié pour traiter l'assembleur embarqué GNU. Bien que non conceptuelle, la principale limitation de cette approche est le coût de développement très important qui les limite ainsi au jeu d'instructions ARMv6. Les travaux contemporains aux nôtres de Corteggiani et al. [CCF18] réutilise les traducteurs de code binaire (vers LLVM), traitant l'assembleur embarqué comme un sous cas d'assembleur standard en compilant le bloc assembleur embarqué – les jetons sont concrétisés, faisant ainsi perdre les informations liées à l'interface.

4.7 Conclusion

Dans ce chapitre, nous avons proposé une représentation intermédiaire et une sémantique de l'assembleur embarqué qui sera utilisée à la fois pour la vérification de l'interface (chapitre 5) et pour la traduction vers le C à des fins de vérification (chapitre 6).

Contrairement aux approches antérieures consistant à concrétiser l'assembleur embarqué dans le but de le traiter comme de l'assembleur standard, notre représentation intermédiaire intègre la notion de jeton, ouvrant la voie aux analyses indépendantes de l'affectation de jetons choisie par le compilateur, et notre sémantique est intermédiaire entre C et binaire.

Nous avons proposé et implémenté une méthode pratique d'extraction de cette représentation intermédiaire, générique et robuste, basée sur une chaîne d'outils existants et obtenant d'excellents résultats en pratique (**85**% des blocs rencontrés dans la distribution Debian *Jessie*). En particulier, cette approche nous permet de supporter les deux jeux d'instructions assembleur les plus courants aujourd'hui, le x86 et l'ARM, et offre des perspectives d'évolution rapide à mesure que de nouveaux jeux d'instructions seront supportés par notre plateforme de désassemblage BINSEC.

Chapitre 5

RustInA : conformité à l'interface

Sommaire

Lors de la compilation d'un bloc assembleur embarqué, le compilateur se base uniquement sur les informations déclarées dans l'interface. Quand ces informations sont inexactes ou incomplètes, de subtiles erreurs peuvent apparaître en fonction des choix du compilateur et de ses passes d'optimisation. La section 2.2 illustre deux défauts de conformité provoquant des bugs dans le programme compilé.

Ce chapitre formalise les propriétés qu'un bloc assembleur embarqué doit respecter pour que son utilisation ne soit pas sujette aux choix du compilateur : *la conformité à l'interface*. Nous proposons ensuite une technique de vérification automatique de ces propriétés. Dans la majorité des cas où une violation est détectée, notre approche est capable de générer un correctif pour mettre à jour l'interface.

Notre approche est ensuite évaluée sur le corpus d'exemples présenté en section 3.2.3, démontrant son efficacité et justifiant son intérêt en mettant en lumière 2183 défauts de conformité impliquant 1364 blocs rencontrés.

La motivation de notre approche, à savoir les risques liés à la non-conformité de l'interface, est discutée a posteriori dans la section 5.6 en se basant sur les résultats expérimentaux ainsi obtenus.

5.1 Définition formelle

Cette section utilise les notations définies en section 4.3.

Intuitivement, un bloc d'assembleur embarqué étendu $\mathtt{X}^{\diamond} \triangleq (\mathtt{C}^{\diamond}, \mathtt{I}^{\diamond})$ est dit conforme à l'interface (*interface compliant*) si et seulement si les déclarations et utilisations des opérandes sont cohérentes – cette condition sera appelée **respect du cadre** (section 5.1.2), et si la sémantique du bloc est indépendante des choix du compilateur – cette condition sera appelée **conditions d'unicité** (section 5.1.3).

Notre formalisation du respect du cadre et de la condition d'unicité repose sur le principe d'équivalence observationnelle que nous allons maintenant détailler.

5.1.1 Équivalence observationnelle

Nous définissons ici une relation d'équivalence observationnelle, notée $\overset{\blacklozenge}{\simeq}{}^{\mathtt{T}_1,\mathtt{T}_2}_{\mathtt{O},\mathtt{F}}$, entre deux états mémoires $\mathtt{S}_1$ et $\mathtt{S}_2$, paramétrée par deux affectations de jetons ($\mathtt{T}_1$ pour $\mathtt{S}_1$ et $\mathtt{T}_2$ pour $\mathtt{S}_2$), un ensemble de jetons observables $\mathtt{O}$: `token set` et un indicateur de séparation mémoire $\mathtt{F}$.

La relation $\overset{\blacklozenge}{\simeq}{}^{\mathtt{T}_1,\mathtt{T}_2}_{\mathtt{O},\mathtt{F}}$ (définition 3) se compose d'une contrainte sur les jetons (définition 1) et d'une contrainte sur la mémoire (définition 2).

Définition 1 *Deux états mémoire S_1 et S_2 sont dits équivalents sur l'ensemble de jetons observés O à travers les affectations de jetons T_1 et T_2, si pour chaque jeton t appartenant à O, l'évaluation de t par T_1 est la même dans S_1 que l'évaluation de t par T_2 dans S_2 :*

$$S_1 \overset{\lozenge}{\simeq}{}^{T_1,T_2}_{O} S_2 \equiv \bigwedge_{t \in O} eval(S_1,\ T_1(t)) = eval(S_2,\ T_2(t))$$

Définition 2 *Deux états mémoire S_1 et S_2 sont équivalents sur la mémoire, si ces deux mémoires sont égales en chaque adresse :*

$$\left(S_1 \triangleq (\_, M_1)\right) \overset{\bullet}{\simeq} \left(S_2 \triangleq (\_, M_2)\right) \equiv M_1 = M_2$$

La relation d'équivalence observationnelle est définie par la conjonction de ces 2 définitions.

Définition 3 *Deux états mémoire S_1 et S_2 sont observationnellement équivalents s'ils sont à la fois équivalents sur l'ensemble de jetons observés O à travers les affectations de jetons T_1 et T_2, et équivalents sur la mémoire :*

$$S_1 \overset{\blacklozenge\,T_1,T_2}{\cong}_{O,F} S_2 \equiv \left(S_1 \overset{\lozenge\,T_1,T_2}{\cong}_{O} S_2\right) \wedge \left(F = \mathit{false} \implies S_1 \overset{\bullet}{\cong} S_2\right).$$

5.1.2 Respect du cadre (*framing condition*)

Le *cadre de l'interface* restreint ce qui peut être lu et ce qui peut être modifié par le bloc. Étant donné une affectation de jetons $\mathtt{T}$, un emplacement est dit *déclaré en lecture* s'il s'agit d'un emplacement associé à un jeton d'entrée (appartenant à $\mathtt{B}^{\mathtt{I}}$). Un emplacement est dit *déclaré en écriture* s'il s'agit d'un emplacement associé à un jeton de sortie (appartenant à $\mathtt{B}^{\mathtt{O}}$) ou d'un registre déclaré en tant que *clobber* (appartenant à $\mathtt{S}^{\mathtt{C}}$). Tout emplacement mémoire est déclaré en lecture et en écriture en l'absence de séparation mémoire ($\mathtt{F} = \mathtt{false}$).

Le *cadre de l'interface* impose alors que :

1. seuls les emplacements déclarés en lecture ou précédemment modifiés peuvent être lus ;
2. seuls les emplacements déclarés en écriture peuvent être modifiés.

Notons qu'il existe une exception au point 2 : il est autorisé de modifier temporairement un emplacement sans permission d'écriture si ce dernier est restauré à sa valeur initiale avant le retour du bloc.

Notons également qu'une règle supplémentaire découle du point 1 : sachant que chaque emplacement associé à un jeton de sortie est considéré comme lu, il doit au moins être soit déclaré en lecture, soit initialisé par le bloc.

Plus formellement, les règles suivantes doivent être respectées :

Définition 4 (*frame-write*) *Pour tout état S, pour toute affectation de jetons T appartenant à S^T, pour tout emplacement l non déclaré en écriture :*

$$S(l) \;=\; \mathit{exec}(S,\; C'\langle T\rangle)(l)$$

Définition 5 (*frame-read*) *Pour tous états S_1 et S_2 et affectation de jetons T appartenant à S^T tel que $S_1 \overset{\blacklozenge\,T}{\cong}_{B^I,F} S_2$:*

$$\mathit{exec}(S_1,\; C'\langle T\rangle) \;\overset{\blacklozenge\,T}{\cong}_{B^O,F}\; \mathit{exec}(S_2,\; C'\langle T\rangle)$$

Documentation GNU. Il arrive qu'un jeton et son expression C associée ne fassent pas la même taille (en bits). Cela se produit par exemple lorsqu'un caractère (`char` 8 bits) est associé au registre `%eax` (32 bits). Certaines architectures comme x86 peuvent alors faire référence uniquement à la sous-partie du registre (e.g. `%al`) là où d'autres comme ARM n'ont pas cette distinction.
Dans le cas d'une entrée, les bits de poids fort sont considérés comme indéfinis et seule la sous-partie correspondant à la taille de l'expression C est déclarée en lecture : en pratique le compilateur peut choisir la méthode d'extension qui l'arrange. Dans le cas d'une sortie, en revanche, le registre entier sera déclaré en écriture et les bits de poids fort pourront être modifiés sans créer une violation.

5.1.3 Condition d'unicité

La condition d'unicité assure que la sémantique du bloc assembleur n'est pas ambigüe au regard du choix de l'affectation de jetons : son comportement doit être le même quel que soit le choix du compilateur.

Une ambiguïté se produit si, parmi les affectations de jetons valides, il existe au moins deux affectations telles que la valeur d'un registre observé n'est pas la même. Pour rappel, ce problème a été illustré dans la section 2.2.2 : si le compilateur choisit le registre `%ebx`, son contenu est écrasé avant d'être utilisé, alors que s'il choisit `%edx`, le programme ne rencontrera pas de *bugs*.

Formellement, un bloc assembleur respecte la condition d'unicité si :

Définition 6 (*unicity*) *Pour tout états S_1 et S_2 et affectations de jetons T_1 et T_2 appartenant à S^T tel que $S_1 \overset{\blacklozenge}{\cong}^{T_1,T_2}_{B^I,F} S_2$:*

$$exec(S_1,\ C'\langle T_1\rangle) \overset{\blacklozenge}{\cong}^{T_1,T_2}_{B^0,F} exec(S_2,\ C'\langle T_2\rangle)$$

À noter que la condition *frame-read* est un cas particulier d'unicité pour lequel $\mathtt{T}_1 = \mathtt{T}_2$. Pour des raisons de compréhension mais aussi d'un point de vue pratique pour la vérification (section 5.2), il reste avantageux de garder ces définitions distinctes.

5.2 Méthode de vérification

Cette section propose une méthode de vérification statique des propriétés de conformité à l'interface. Elle s'appuie sur des techniques d'analyses de flot de données (*dataflow analysis framework* [Kil73]) largement utilisées pour les optimisations dans les compilateurs et les outils de vérification de logiciels. L'approche consiste à collecter un ensemble de faits (*dataflow facts*) sur les emplacements (registres, mémoire et jetons) et à les comparer aux faits nominaux déduits de la lecture de l'interface (section 3.2.2).

5.2.1 Respect du cadre

Vérifier que le cadre est respecté requiert de collecter les informations classiquement nommées ***use-def***, c'est à dire les liens entre les valeurs lues et écrites par le code. Nous utilisons deux analyses de flot de données (section 3.1.3) :

— une analyse de flots de données en avant pour collecter les emplacements modifié par le bloc ;

— une analyse de vivacité (*liveness analysis*) pour collecter les variables utiles (*alive*) en chaque point de contrôle du bloc.

Les deux analyses de flots de données reposent sur l'algorithme général d'analyse de flots décrit dans la section 3.1.3. La définition des différentes fonctions sera détaillée pour chacun des cas.

Dans les deux cas, la mémoire est considérée comme un tout : l'analyse conclura que la mémoire est modifiée dès lors qu'une écriture en mémoire a lieu et que la mémoire est lue dès lors qu'un pointeur est déréférencé. Cette sur-approximation présente un certain nombre d'avantages : d'un part, elle correspond au mode de fonctionnement de l'interface concrète qui considère la mémoire dans son ensemble, et d'autre part elle facilite les analyses de flots de données en réduisant la taille des ensembles d'emplacements à considérer, retirant la nécessité d'analyse statique sur la mémoire (*heap, points-to*) et accélérant ainsi la convergence de l'analyse.

Nous allons présenter une première version des analyses de flot de données en section 5.2.1.1 puis nous détaillerons en section 5.2.1.2 deux optimisations permettant de réduire le nombre de faux positifs liées aux sur-approximations.

5.2.1.1 Analyses basiques

frame-write. L'analyse doit garantir que chaque emplacement non déclaré en écriture possède la même valeur à la fin de l'exécution que celle qu'il possédait au début de l'exécution.

En première approximation, un emplacement qui ne fait jamais partie du membre gauche d'une affectation (suffisant du fait que la représentation intermédiaire est sans effet de bord), aura conservé sa valeur initiale. L'étape principale consiste ainsi à vérifier que chaque emplacement apparaissant à gauche d'une affectation appartient à l'ensemble des emplacements déclarés en écriture ($\texttt{B}^{\texttt{0}} \bigcup \texttt{S}^{\texttt{C}}$ avec la mémoire si `F = false`).

La définition de l'analyse de flots de données est donnée dans le flot de données 2.

frame-read. L'analyse doit garantir que chaque emplacement lu a été correctement initialisé au préalable.

L'analyse arrière classique de calcul de vivacité (*backward liveness analysis*) permet d'obtenir une sur-approximation des emplacements qui seront lus à ou après un point de contrôle donné : en partant de la sortie du bloc, elle remonte en arrière en calculant les dépendances (membre de droite d'une affectation ou condition de

Flot de données 2: Collecte des définitions

```
type data : location set
define StartNode(cfg) = cfg.entryNode
define SuccEdges(node) = node.forwardEdges
define Init(node) = ∅
define Transfer(edge, data) =
    match edge.instr with
        case v ← _ : data ∪ { v }  // register or token
        case @[_]_ ← _ : data ∪ { memory }  // store
        otherwise : data  // branches
define Join(data, data') = data ∪ data'
```

branchement). Lorsque le point fixe est atteint, l'analyse vérifie que tous les emplacements lus à partir de l'entrée du bloc appartiennent à l'ensemble des emplacements déclarés en lecture ($\mathtt{B}^{\mathtt{I}}$ avec la mémoire si $\neg\mathtt{F}$).

Le calcul de dépendances d'une expression dans sa version basique est donné dans la fonction Deps ci-dessous.

Function Deps(expr) : Calcul de dépendances (basique)

```
match expr with
    case v : { v }  // register or token
    case @[a]_ : Deps(a) ∪ { memory }  // load
    case _ x : Deps(x)  // unary
    case x _ y : Deps(x) ∪ Deps(y)  // binary
    case x ? y : z : // ternary
     | Deps(x) ∪ Deps(y) ∪ Deps(z)
    otherwise : ∅  // constant
```

La configuration de l'analyse de flots de données est présentée dans le pseudo-code 3 où la variable `Outputs` est l'ensemble $\mathtt{B}^{\mathtt{O}}$ des jetons déclarés en sortie.

5.2.1.2 Amélioration de la précision

Nous proposons ici deux optimisation A (*frame-write*) et B (*frame-write*) pour réduire le nombre de faux positifs liées aux sur-approximations.

Optimisation A (*propagation d'expressions*). La technique basique se base sur le fait qu'un emplacement qui n'apparaît pas dans le membre gauche d'une affection conserve sa valeur d'origine. Cette approximation donne de bons résultats mais peut donner lieu à de faux positifs lorsque un emplacement est modifié temporairement avant d'être restauré à sa valeur d'origine. C'est le cas des registres sauvegardés dans la pile puis dépilés avant la fin du bloc. D'autres cas, tels que l'incrément temporaire d'un pointeur suivi d'un décrément, conservent la valeur de l'emplacement même s'il apparaît dans une affectation.

Pour éviter cette source de faux positifs, l'analyse propage symboliquement les expressions (*inlining*) tout en réalisant des simplifications syntaxiques à l'aide de

Flot de données 3: Collecte des utilisations

```
type data : location set
define StartNode(cfg) = cfg.exitNode
define SuccEdges(node) = node.backwardEdges
define Init(node) = if node.isExit then Outputs else ∅
define Transfer(edge, data) =
    match edge.instr with
        case v ← x when v ∈ data : // live
          | ( data \ {v} ) ∪ Deps(x)
        case @[a] _ ← x : // store
          | data ∪ Deps(a) ∪ Deps(x)
        case if x then goto _ else goto _ :
          | data ∪ Deps(x)
        otherwise : data
define Join(data, data') = data ∪ data'
```

règles de réécriture (détaillées en annexe B). Lors de la vérification de la propriété *frame-write*, chaque candidat potentiel de violation obtenu par la première méthode est réévaluer et aucune alarme n'est levée si le résultat de la propagation symbolique est l'identité.

Le but est donc ici de reconnaître si une affectation correspond à la fonction identité. Le pseudo-code 4 ci-dessous donne la définition de l'analyse de flot de données correspondante.

Flot de données 4: Propagation d'expressions

```
type lattice : ⊥ | expr | ⊤
type data : location ↦ lattice
define StartNode(cfg) = cfg.entryNode
define SuccEdges(node) = node.forwardEdges
define Init(node) = default ⊥
define Transfer(edge, data) =
    match edge.instr with
        case v ← e : data[v ↦ Eval(data, e)] // registre ou jeton
        case @[_] _ ← _ : Filter(data, hasLoad) // écriture en mémoire
        otherwise : data // branches
define Join(data, data') =
    foreach lvalue do
        match data[lvalue], data'[lvalue] with
            case ⊥, ⊥ : result[lvalue ↦ ⊥]
            case e, e : result[lvalue ↦ e]
            otherwise : result[lvalue ↦ ⊤]
define Filter(data, f) =
    foreach lvalue do
        match data[lvalue] with
            case ⊥ : result[lvalue ↦ ⊥]
            case e : if f(e) then result[lvalue ↦ ⊤] else result[lvalue ↦ e]
            case ⊤ : result[lvalue ↦ ⊤]
```

Dans la fonction de transfert, à chaque affectation, le membre de droite est évalué

dans l'environnement calculé par l'analyse. Si l'expression n'a qu'une seule valeur abstraite possible, la fonction d'évaluation Eval renvoie cette expression. Sinon, elle renvoie $\top$ (voir ci-après). Les expressions sont simplifiées syntaxiquement à la volée, en particulier les opérations réversibles telle que l'addition, la soustraction ou la disjonction exclusive (« *xor* »).

Function Eval(env, expr) : Évaluation d'expressions

```
match expr with
| case v : // register or token
|   | match data[v] with
|   |   | case ⊥ : v  // input
|   |   | case e : e
|   |   | case ⊤ : ⊤
| case @[x]_n : // load
|   | match Eval(env, x) with
|   |   | case e : @[e]_n
|   |   | otherwise : ⊤
| case unop x : // unary
|   | match Eval(env, x) with
|   |   | case e : unop x
|   |   | otherwise : ⊤
| case x binop y : // binary
|   | match Eval(env, x), Eval(env, y) with
|   |   | case e1, e2 : e1 binop e2
|   |   | otherwise : ⊤
| case x ? y : z : // ternary
|   | match Eval(env, x), Eval(env, y), Eval(env, z) with
|   |   | case true, e, _ : e
|   |   | case false, _, e : e
|   |   | case e1, e2, e3 : e1 ? e2 : e3
|   |   | otherwise : ⊤
| otherwise : expr  // constant
```

Sans information sur la mémoire et l'*aliasing* de pointeurs, il n'est pas possible traiter les écritures mémoire précisément. La fonction de filtrage `Filter` permet « d'oublier » (mettre à $\top$) toutes les expressions qui dépendent d'une lecture en mémoire. La fonction hasLoad renvoie vrai (`true`) lorsque une expression dépend d'une lecture en mémoire.

Function hasLoad(expr) : calcul de dépendances sur la mémoire

```
match expr with
| case @[_]_ : true  // load
| case _ x : hasLoad(x)  // unary
| case x _ y : hasLoad(x) ∪ hasLoad(y)  // binary
| case x ? y : z : // ternary
|   | hasLoad(x) ∪ hasLoad(y) ∪ hasLoad(z)
| otherwise : false  // constant, register or token
```

Optimisation B (*granularité niveau bit*). Au niveau source, le calcul de vivacité se fait généralement au niveau des variables. Sur du code bas niveau, les « variables » (registres) peuvent être adressées au niveau de l'octet ou du bit – certains registres sont même spécialisés pour contenir plusieurs entités logiques différentes. Considérer les dépendances au niveau d'un registre peut ainsi entraîner une perte de précision, pouvant mener à de fausses alarmes. Pour éviter cette perte de précision, notre analyse suit précisément le statut de chacun des bits des registres. La fonction Deps' décrite ci-après présente conceptuellement le nouveau calcul de dépendances.

Function Deps'(expr, bit) : calcul de dépendances niveau bit

match expr **with**

case v **:** { (v, bit) } // register or token

case @[a]$_{-}$ **:** $\bigcup_{i=0}^{\texttt{sizeof}(a)-1}$ Deps'(a, i) ∪ { memory } // load

case $\neg x$ **:** Deps'(x, bit)

case $-x$ **:** $\bigcup_{i=0}^{\texttt{bit}}$ Deps'(x, i)

case $\texttt{zext}_n$ x **:** **if** bit < n **then** Deps'(x, bit) **else** ∅

case $\texttt{sext}_n$ x **:** **if** bit < n **then** Deps'(x, bit) **else** Deps'(x, *sizeof*(x)−1)

case $x_{\{j..i\}}$ **:** **if** $i \leq$ bit $\leq j$ **then** Deps'(x, bit) **else** ∅ // restrict

case $x \wedge y \mid x \vee y \mid x \oplus y$ **:**
| Deps'(x, bit) ∪ Deps'(y, bit)

case $x + y \mid x - y \mid x \times y$ **:**
| $\bigcup_{i=0}^{\texttt{bit}}$ Deps'(x, i) ∪ $\bigcup_{i=0}^{\texttt{bit}}$ Deps'(y, i)

case x :: y **:** **if** bit < *sizeof*(y) **then** Deps'(y, bit) **else** Deps'(x, bit − *sizeof*(y))

case x _ y **:** $\bigcup_{i=0}^{\texttt{sizeof}(x)-1}$ Deps'(x, i) ∪ $\bigcup_{i=0}^{\texttt{sizeof}(y)-1}$ Deps'(y, i) // binary

case x ? y : z **:** Deps'(x, 0) ∪ Deps'(y, bit) ∪ Deps'(z, bit) // ternary

otherwise : ∅ // constant

Ce nouvel algorithme exploite 3 situations où la fonction Deps renvoie une surapproximation :

— les bits en dehors de la plage de l'opérateur de restriction n'influencent pas le résultat ;

— les bits du résultat d'une opération bit-à-bit (*bitwise*) sont indépendants ;

— le résultat tronqué de l'addition, de la soustraction ou de la multiplication ne dépend que des bits de poids faible de ces arguments.

En pratique, nous utilisons une structure de donnée plus compacte pour représenter l'ensemble des paires (variable, position du bit vivant). La structure se

présente sous la forme d'une association entre une variable et un vecteur de bits (*bitset*) de la même taille que la variable et dont les bits à `1` signifie que le bit est vivant. L'union et l'intersection se font très simplement à l'aide du « ou bit-à-bit » et du « et bit-à-bit ».

5.2.2 Condition d'unicité

Vérifier que le bloc est sûr au regard de la condition d'unicité consiste à s'assurer que quelle que soit l'allocation valide de jetons sélectionnée, le chemin de données au sein du bloc est le même.

Au vu de la nature particulière du problème (variants de code isomorphes), si tous les emplacements lus sont correctement initialisés (*frame-read*), alors le seul cas pathologique pouvant apparaître est qu'un emplacement utilisé dans le futur se fasse « capturer » (dans le sens de la substitution syntaxique) par une écriture dans un autre emplacement. Il y a alors un problème d'unicité si la « capture » se produit pour certaines affectations de jetons mais pas pour toutes.

Afin de vérifier que ce cas ne se produit pas, l'analyse s'appuie sur la relation `may_impact` entre deux emplacements $\texttt{lvalue}^\blacklozenge$ (registre, mémoire ou jeton). Cette relation est définie tel que `l may_impact l'` est faux s'il est possible de prouver qu'écrire dans `l` ne peut pas avoir d'impact sur l'évaluation futur de `l'` et ce, quelque soit l'affectation de jeton. Dans l'idéal, `l may_impact l'` doit répondre faux (`false`) s'il n'existe aucune affectation de jeton telle que `l` devienne une sous-expression de `l'`.

À l'aide de cette relation et des informations précédemment obtenues par les analyses de *frame-write* et *frame-read*, l'analyse vérifie à chaque point de contrôle qu'il n'existe pas d'emplacement `l'` lu dans le futur (*alive*) tel que pour l'affectation à l'emplacement `l`, la propriété `l may_impact l'` renvoie vrai (`true`).

Calcul effectif de `may_impact`. L'objectif derrière `may_impact` est de ne pas devoir énumérer l'ensemble d'affectations de jetons valides $\mathtt{S}^\mathtt{T}$ dont le cardinal peut être élevé. Pour cela, l'ensemble $\mathtt{S}^\mathtt{T}$ est approximé par une abstraction où les emplacements $\texttt{lvalue}^\blacklozenge$ sont réduits à des emplacements abstraits $\texttt{lvalue}^*$. Un emplacement abstrait $\texttt{lvalue}^*$ indique seulement si l'emplacement est une valeur constante (`Immediate`), un registre (`Direct variable`) ou est utilisé pour calculer une adresse d'un accès mémoire (`Indirect variable`). L'affectation de jetons est abstraite en réinterprétant les contraintes de la syntaxe concrète dans le domaine des emplacements abstraits, donnant l'affectation $\mathbb{D}^* : \texttt{lvalue}^\blacklozenge \mapsto \texttt{lvalue}^* \ \texttt{set}$.

L'effet d'une écriture dans un emplacement abstrait l^* sur un emplacement abstrait $l^{*\prime}$ est approximé par la relation $l^* \ \texttt{impact}^* \ l^{*\prime}$ et est défini comme :

$$l^* \ \texttt{impact}^* \ l^{*\prime} = \begin{cases} \texttt{true} & \text{for } \texttt{Direct } r \ \texttt{impact}^* \ \texttt{Direct } r \\ \texttt{true} & \text{for } \texttt{Direct } r \ \texttt{impact}^* \ \texttt{Indirect } r \\ \texttt{false} & \text{otherwise} \end{cases}$$

La relation `l may_impact l'` est alors construite comme retournant vrai (`true`) (sur-approximation) sauf dans les cas particuliers suivants :

Disjoint : il n'existe pas de paire l^*, $l^{*\prime}$ appartenant à $\mathbb{D}^*(\texttt{l}) \times \mathbb{D}^*(\texttt{l'})$ tel que l^* $\texttt{impact}^*$ $l^{*\prime}$;

Clobber au moins un des emplacements `l` ou `l'` est déclaré comme *clobber* ($\texttt{S}^{\texttt{C}}$ disjoint par définition) ;

Early clobber les deux emplacements `l` et `l'` sont des jetons et `l` est déclaré comme étant *early clobber* (`"&"` disjoint des autres jetons par définition) ;

Invariant `l` et `l'` sont égaux quelle que soit l'affectation de jetons (contrainte d'égalité tel que `"0"`).

Nous proposons une méthode basée sur l'analyse de flot de données pour vérifier chacune des conditions nécessaires à la conformité de l'interface : le *frame-read*, le *frame-write* et l'unicité. Cette méthode est correcte et il est ainsi possible de prouver la conformité de l'interface des blocs assembleur embarqué analysés. Cependant, les sur-approximations peuvent mener à des fausses alarmes et nous proposons deux optimisations, la propagation d'expression symbolique et le calcul de dépendance de niveau bit pour en réduire le nombre – nos expérimentations montrent que les faux positifs ont presque entièrement disparu dans notre corpus d'étude (section 5.5.1.4).

5.3 Correction automatique des erreurs de conformité

Quand l'analyse détecte un problème de conformité entre le code assembleur $\texttt{C}^{\diamond}$ et l'interface $\texttt{I}^{\diamond}$, il est souvent possible de générer un correctif en modifiant uniquement les déclarations de l'interface $\texttt{I}^{\diamond}$. Les correctifs sont détaillés dans les sections suivantes en fonction de la nature de la violation détectée.

Notons que la détection étant basée sur une sur-approximation, le risque de faux-positif existe (discuté en section 5.5.1.4). Ceci dit, les correctifs peuvent tout de même être générés et, si le bloc assembleur reste compilable (si $\texttt{S}^{\texttt{T}} \neq \emptyset$), le bloc restera fonctionnellement équivalent. Il pourra néanmoins être compilé vers un code moins efficace (discuté en section 5.4).

5.3.1 Correctifs pour le respect du cadre

Un correctif est généré pour ajouter une déclaration à l'interface sur un emplacement mal utilisé.

frame-write. Si un registre « écrit en dur » dans le code est modifié, il est ajouté à l'ensemble des *clobber* $\texttt{S}^{\texttt{C}}$. Si un jeton est modifié sans être déclaré en écriture, il est ajouté à l'ensemble des sorties $\texttt{B}^{\texttt{O}}$.

frame-read. Si un registre « écrit en dur » dans le code est lu sans avoir été préalablement initialisé, la génération de correctif échoue et le problème est remonté comme une erreur grave. Si un jeton est lu sans avoir été initialisé au préalable (soit par écriture, soit étant déclaré en lecture), il est ajouté à l'ensemble des entrées $\texttt{B}^{\texttt{I}}$.

Dans les deux cas, un accès mémoire alors que les mémoires sont séparées ($\texttt{F} = \texttt{true}$) retire la séparation mémoire : $\texttt{F} = \texttt{false}$.

Les changements sur l'interface formelle $\texttt{I}^{\diamond}$ peuvent ensuite être répercutés sur la syntaxe concrète. La particularité de la syntaxe GNU est que le patron (*template*) fait référence aux élements de la liste de façon « positionnelle ». Il est donc important lors de l'ajout ou la suppression d'élements de maintenir la numérotation des jetons à jour dans le patron.

5.3.2 Correctifs pour la condition d'unicité

Un problème d'unicité n'est généralement pas anodin et le corriger de façon satisfaisante requiert la plupart du temps l'intervention d'un humain. En effet, le bloc fautif possède plus d'une interprétation et il est préférable d'en identifier la cause pour ne garder que celle initialement prévue par le développeur.

Dans certain cas, résoudre un problème d'unicité en ne corrigeant que l'interface forcerait à sur-contraindre chacun des éléments présents dans l'interface en retirant tout choix au compilateur. Il est parfois préférable de repenser le bloc et ainsi de modifier et l'interface, et le patron pour résoudre le problème. De telles modifications ont par exemple été retrouvées dans l'historique du projet libatomic_ops (commit 64d81cd).

Toutefois, cela n'empêche pas la suggestion automatique d'un correctif pouvant résoudre le problème. En l'occurrence, lorsqu'un registre « écrit en dur » est l'origine du problème, il est ajouté à l'ensemble des *clobber* $\texttt{S}^{\texttt{C}}$ et ce, même s'il ne viole pas la propriété *frame-write* . Lorsqu'un jeton est l'origine du problème, il est ajouté à la liste des *early clobbers* (déclaré avec le modificateur `"&"`).

Notons que la syntaxe concrète ne permet pas de déclarer facilement une inégalité entre deux jetons ou un jeton et un registre, ce qui force à utiliser une inégalité globale (*clobber* & *early clobber*) et limite l'efficacité de la solution.

5.4 Raffinement automatique de l'interface

Lorsqu'une interface est « incomplète », des problèmes de compilation peuvent apparaître et l'utilisation du bloc est incertaine. Nous allons maintenant discuter du cas opposé où une interface serait plus contrainte que « nécessaire ».

Nous considérons ici qu'une entrée dans l'interface « n'est pas nécessaire » si le fait de la retirer ne change pas le comportement ni la propriété de conformité du bloc.

Raffinement des entrées La technique de vérification de la conformité de l'interface repose sur l'inférence d'une sur-approximation de l'interface nécessaire pour garantir la propriété. De ce fait, si une entrée est présente dans l'interface en cours d'évaluation mais absente de celle inférée, il est assuré qu'elle n'est pas nécessaire et peut être supprimée en toute sécurité.

En bonus à la vérification, il est donc possible de supprimer des registres déclarés en *clobber* ou des jetons déclarés en écriture sans jamais être modifiés ou des jetons déclarés en lecture sans jamais être lus.

Il est néanmoins possible que des éléments ne participant pas à la production du résultat, et donc jugés non nécessaires, aient été ajoutés intentionnellement dans l'objectif d'empêcher localement le compilateur de réaliser des optimisations. Dans ce cas, il n'est pas souhaitable de raffiner l'interface automatiquement. N'étant pas un élement crucial pour la sûreté, les normalisations opérées par notre prototype ne sont que suggérées par défaut.

Ces raffinements sont toutefois modestes et n'auront que peu d'impact sur l'exécution du programme final. En revanche, les développeurs d'assembleur essaieront de supprimer en priorité le mot clé `"memory"` qui a un impact généralement important sur la production de code. En effet, `"memory"` inhibe les optimisations sur les variables placées en mémoire, même si elles n'ont pas de lien avec le bloc assembleur.

Raffinement de la mémoire Le mot clé `"memory"` est requis dès lors qu'un accès mémoire arbitraire est réalisé par le bloc. La documentation GNU propose toutefois des alternatives si les accès mémoire sont limités à des déplacement linéaire de pointeurs par rapport à la valeur de jetons.

L'exemple ainsi donné est celui d'une fonction itérant sur une chaîne de caractères. Les accès mémoire sont bornés au bloc mémoire associé au pointeur donné en paramètre. Différentes syntaxes sont alors possibles pour donner plus de liberté au compilateur. Le bloc original utilisant le mot clé `"memory"` est donné en figure 5.1.

```
int strlen (char *p)
{
  int count;
  __asm__ ("repne scasb"
           : "=c" (count), "+D" (p)
           : "0" (-1), "a" (0), "memory");
  return -2 - count;
}
```

FIGURE 5.1 – Utilisation classique de `"memory"`

Si les chaînes de caractères passées en paramètre sont limitées à une longueur maximale de caractères, par exemple, un tampon de taille fixe de 10 caractères, il est possible de remplacer le mot clé `"memory"` par une entrée mémoire. Le pointeur d'entrée `p` est réinterprété (*cast*) comme un pointeur vers un type de taille 10. La figure 5.2 présente la solution passant par une structure `struct` et la figure 5.3 présente une alternative passant par un tableau de taille constant.

```
int strlen (char *p)
{
  int count;
  __asm__ ("repne scasb"
           : "=c" (count), "+D" (p)
           : "m" (*(const struct { char k[10]; } *) p), "0" (-1), "a" (0));
  return -2 - count;
}
```

FIGURE 5.2 – Utilisation d'une `struct` de taille finie

```
int strlen (char *p)
{
  int count;
  __asm__ ("repne scasb"
           : "=c" (count), "+D" (p)
           : "m" (*(const char (*)[10]) p), "0" (-1), "a" (0));
  return -2 - count;
}
```

FIGURE 5.3 – Utilisation d'un tableau de taille finie

Finalement, si la taille du bloc mémoire n'est pas connue à l'avance, il est possible de réinterpréter le pointeur vers un type incomplet, comme illustré en figure 5.4. Ici, le compilateur est informé que la totalité du bloc mémoire adressé par `p` peut être lue. Le compilateur ne devra ici être conservateur que sur la mémoire pouvant « *aliaser* » avec le pointer `p`, ce qui est une amélioration par rapport au mot clé `"memory"`.

```
int strlen (char *p)
{
  int count;
  __asm__ ("repne scasb"
           : "=c" (count), "+D" (p)
           : "m" (*(const char (*)[]) p), "0" (-1), "a" (0));
  return -2 - count;
}
```

FIGURE 5.4 – Utilisation d'un tableau non borné

Documentation GNU. Bien que présenté et illustré dans la documentation, le traitement de cette syntaxe est inégal au sein des compilateurs. Ainsi si GCC supporte ces différentes façons de spécifier un accès en mémoire, Clang refuse les entrées impliquant des types incomplets (tableaux et structures de tailles dynamiques). Pour les raisons évoquées en section 4.3.3, ICC ne supporte pas de la façon attendue les entrée impliquant un tableau. Ces solutions sont donc à éviter pour maximiser la portabilité du code entre les compilateurs.

Notre objectif est ici de déterminer automatiquement si le mot clé `"memory"` peut être remplacé par des entrées de type mémoire. Il s'agit donc de décider s'il n'y a pas d'accès mémoire arbitraires, et de calculer l'ensemble des jetons utilisés comme bases de pointeurs ainsi que la taille des blocs mémoire associés. Pour cela, nous modifions

légèrement l'algorithme de propagation d'expression présenté en section 5.2.1.

Le nouveau treillis de données intègre un élément supplémentaire `linear`(t, n) où t est un jeton et n est un entier positif ou $\top$:

$$\texttt{upperbound} := \mathit{positive} \mid \top$$
$$\texttt{lattice} := \bot \mid \mathit{expr} \mid \texttt{linear}(\mathit{token} \times \texttt{upperbound}) \mid \top$$

L'élément `linear` permet de représenter un ensemble d'expressions dont la base est un jeton plus un déplacement. Il permet de rester précis lorsque le déplacement est borné par une constante.

La fonction d'évaluation Eval (page 70) a deux cas spéciaux supplémentaires pour l'opérateur **+** afin de produire des abstractions linéaires à la place de $\top$:

Function Eval(env, expr) : Cas particuliers de l'arithmétique

```
case x + y :
    match Eval(env, x), Eval(env, y) with
        case e1, e2 : e1 + e2
        case t, ⊤ : linear(t, ⊤)
        case linear(t, n), c : linear(t, n + c)  // constant
        case linear(t, _), _ : linear(t, ⊤)  // others
        otherwise : ⊤
```

La fonction Join est également mise à jour pour produire et propager les abstractions linéaires. Contrairement à de l'interprétation abstraite qui se veut précise, la convergence est recherchée ici en priorité. Ainsi, à la moindre divergence, la limite d'octets du bloc est fixé à $\top$ (*widening*).

Function Join(data, data') : Cas particuliers de l'arithmétique

```
foreach lvalue do
    match data[lvalue], data'[lvalue] with
        case ⊥, ⊥ : result[lvalue] ← ⊥
        case e, e : result[lvalue] ← e
        case t + c, t + c' : result[lvalue] ← linear(t, max(c, c'))  // constant
        case t + e, t + e' : result[lvalue] ← linear(t, ⊤)  // arbitrary
        case linear(t, _), t + e | t + e, linear(t, _) : // arbitrary
         | result[lvalue] ← linear(t, ⊤)
        case linear(t, n), linear(t, n) : // unchanged
         | result[lvalue] ← linear(t, n)
        case linear(t, _), linear(t, _) : result[lvalue] ← linear(t, ⊤)
        otherwise : result[lvalue] ← ⊤
end
```

Cette nouvelle propagation d'expressions permet de calculer un ensemble de jetons utilisés en tant que bases pour accéder à la mémoire.

La valeur de chaque adresse utilisée pour accéder à la mémoire est évaluée dans la nouvelle abstraction. Il s'agit d'une sur-approximation de l'ensemble des accès mémoire du bloc.

Si cet ensemble contient $\top$ ou n'importe quelle expression qui n'est pas sous la forme « jeton + déplacement », le mot clé `"memory"` est jugé nécessaire. Cela inclut les faux positifs qui seraient dûs aux imprécisions de l'analyse.

En revanche, dans les autres cas, il est possible de transformer l'interface pour retirer le mot clé `"memory"`. Pour chaque jeton apparaissant dans l'ensemble calculé, la taille du bloc accédé est agrégée en prenant le maximum. Dans le cas où la taille du bloc inférée est bornée par une constante, une nouvelle entrée mémoire est ajoutée en suivant la syntaxe de l'une des figures 5.2 et 5.3. Sinon, dans le cas où elle est évaluée à $\top$, la syntaxe de la figure 5.4 est utilisée.

5.5 Evaluation expérimentale de RUSTInA

La vérification de la conformité à l'interface, de la génération de correctifs et du raffinement d'interface ont été implémentés dans notre prototype TInA – dénommé *Repair User Specification by* TInA (RUSTInA).

Le code des algorithmes de détection et de correction de problèmes ajoute environ 1kLOC au code d'extraction de TInA pour un total de 3.5kLOC.

L'évaluation de RUSTInA est faite à travers sept questions de recherche :

RQ1. RUSTInA est-il capable de détecter automatiquement des problèmes de conformité d'interface dans des blocs assembleur rencontrés dans du code réel ?

RQ2. Le cas échéant, combien de problèmes de conformité d'interface sont détectés sur le corpus d'étude (section 3.2.3) et quels sont ceux récurrents ?

RQ3. Les choix de conception influencent-ils le nombre d'erreurs détectées et la proportion de fausses alarmes levées ?

RQ4. RUSTInA est-il capable de corriger automatiquement les problèmes de conformité qu'il remonte ?

RQ5. Le cas échéant, les correctifs ont-ils la même qualité que ceux produits par un développeur ?

RQ6. RUSTInA est-il capable de raffiner automatiquement une interface sur-contrainte ?

RQ7. Quels changements notre prototype a-t-il permis d'apporter sur les projets étudiés ?

5.5.1 Détection de problèmes de conformité (RQ1, RQ2, RQ3)

RUSTInA analyse les 202 projets x86 étudiés et les 3 projets pour ARM (présentés en section section 3.2.3) en moins de 2 minutes (*40 ms par bloc en moyenne*).

Les résultats de la détection sont illustrés à l'échelle des projets dans le tableau 5.1, à l'échelle des blocs dans la tableau 5.2 et enfin à l'échelle des violations individuelles dans la tableau 5.3.

La classification entre violations bénines ou dangereuses est justifiée par notre analyse qualitative des erreurs les plus répandues, discutée en section 5.6.

5.5.1.1 Résultats niveau projet

Corpus x86. Sur les 202 projets, RUSTInA vérifie que 117 projets ne contiennent aucun bloc violant la propriété de conformité, soit une bonne moitié des projets étudiés. Il détecte pour 31 projets (soit environ 15% du total) au moins une violation concernant le registre de condition. Il rapporte pour 54 projets (soit plus du quart) une violation jugée dangereuse.

Corpus ARM. Sur les 3 projets étudiés, un seul ne contient aucun bloc violant la propriété de conformité. Les deux autres contiennent une violation dangereuse. Nous verrons en section 5.5.1.4 qu'un de ces projets est en réalité conforme et qu'il s'agit d'une fausse alarme.

Table 5.1 – Résumé de la détection niveau projet

	x86		ARM	
Projets observés	**202**		3	
– conformes	117	58%	1	33%
– avec problème bénin	31	15%	0	0%
– avec violation dangereuse	54	27%	2[1]	67%

[1] 1 projet contient de fausses alarmes.

5.5.1.2 Résultats niveau bloc

Corpus x86. RUSTInA vérifie que 1292 blocs respectent la propriété de conformité, soit un peu moins de la moitié des blocs observés. Il détecte pour 1070 blocs (soit environ 40% du total) une violation bénine et pour 294 blocs (soit environ 10%) une violation jugée dangereuse.

Corpus ARM. RUSTInA vérifie que 311 blocs respectent la propriété de conformité. Il détecte pour 81 blocs (soit environ 20% du total) une violation jugée dangereuse. Deux des blocs détectés relèvent d'une imprécision de l'analyse.

5.5.1.3 Analyse des violations individuelles

Le détail des violations détectées est donné dans la tableau 5.3. Nous résumons ici la situation.

Table 5.2 – Résumé de la détection niveau bloc

	x86		ARM	
Bloc observés	**2656**		392	
– conformes	1292	49%	311	79%
– avec problème bénin	1070	40%	0	0%
– avec violation dangereuse	294	11%	81[1]	21%

[1] Deux blocs, issus du même projet, provoquent une fausses alarme.

Corpus x86. RUSTInA détecte 2183 problèmes de conformité dont environ la moitié est jugée dangereuse. La grande majorité des problèmes, environ 80%, vient d'une écriture non déclarée (*frame-write*) alors qu'environ 20% des problèmes sont dûs à une lecture non déclarée (*frame-read*). Les problèmes d'unicité sont plus rares – environ 4% du total.

Corpus ARM. RUSTInA détecte 81 problèmes de conformité. L'architecture ARM ne recevant pas le même traitement que le x86 de la part des compilateurs, même les écritures dans le registre de conditions sont jugées dangereuses (voir section 5.6). La grande majorité des problèmes, 96%, vient d'une écriture non déclarée (*frame-write*) ; le reste des problèmes étant dû à une lecture non déclarée (*frame-read*).

Table 5.3 – Résumé de la détection des violations

	x86		ARM	
Violations détectées	**2183**		81	
violations dangereuses	986	45%	81	100%
frame-write	**1718**	79%	78	96%
– écriture du registre de conditions	1197	55%	78	96%
– écriture d'une entrée en lecture seule	17	1%	0	0%
– écriture d'un registre non déclaré	436	20%	0	0%
– écriture en mémoire non déclarée	68	3%	0	0%
frame-read	**379**	17%	3	4%
– lecture d'une entrée en écriture seule	19	1%	2[1]	2%
– lecture d'un registre non déclaré	183	8%	0	0%
– lecture en mémoire non déclarée	177	8%	1	1%
unicity	**86**	4%	0	0%

[1] Fausses alarmes causées par les sur-approximation de l'analyse.

5.5.1.4 Évaluation des violation reportées

Les blocs jugés conformes par RUSTInA sont évidemment supposés l'être, mais ce peut être utile à vérifier vis-à-vis du prototype. En revanche une violation détectée peut, comme expliqué en section 5.2, n'être qu'une fausse alarme.

Pour évaluer ces deux cas, nous avons recours à un examen manuel des violations reportées sur le corpus x86 :

— sur les exemples présentés en section 2.2 ;

— sur les blocs de 8 projets sélectionnés pour la présence importante d'assembleur embarqué dans leurs sources et couvrant plus de la moitié des violations identifiées jugées dangereuse ;

— sur un petit échantillon de 100 blocs choisis au hasard.

Cet examen a permis de révéler deux sources d'imprécision de l'analyse menant à de fausses alarmes. Après identification et conception des optimisation A (propagation d'expression) et B (analyse de vivacité niveau bit) (section 5.2), le rejeu avec et sans optimisations permet de mesurer leur impact : 119 violations jugées dangereuses peuvent être évitées en utilisant conjointement A et B. Les résultats du tableau 5.3 prennent déjà en compte ces améliorations. Ces deux optimisations se comportent de façon indépendante sur nos exemples, A étant responsable de l'élimination de 32 fausses alarmes sur *frame-write* alors que B élimine 87 fausses alarmes sur *frame-read*, soit une diminution de respectivement 6% et 20% des violations reportées.

Un examen similaire sur les 81 violations relevées nous a permis de détecter deux fausses alarmes qui sont dûes au fait que l'analyse n'est pas sensible au chemin (*path insensitive*).

Les examens manuels n'ont pas remonté de cas de faux négatifs [1].

Conclusion RQ1, RQ2 et RQ3

Le prototype est applicable sur un grand nombre de blocs tirés d'exemples réels et ce, pour deux architectures différentes. RUSTInA est efficace dans la détection d'un grand nombre de violations de la propriété de conformité à l'interface, ceci avec un taux de fausse alarmes observées proche de zéro.

Les erreurs rencontrées les plus courantes sont dues à des écritures non déclarées (environ 80%) ou des lectures non déclarées (environ 20%).

5.5.2 Génération de correctifs (RQ4)

La technique présentée en section 5.3 est capable de générer un correctif pour tous les types de violations à l'exception du cas de la lecture d'un registre non déclaré (il n'est pas possible d'inférer automatiquement une expression C à lier au registre). Un correctif est donc généré pour chacune des 2000 violations détectées restantes du corpus x86 ainsi que pour les 81 violations du corpus ARM.

1. Impossible en théorie, un faux négatif serait le signe d'une erreur dans l'implémentation.

Afin de vérifier la validité du correctif, la technique de détection est une nouvelle fois appliquée sur les blocs obtenus après l'application des corrections.

Les résultats obtenus sont ceux attendus et chacune des violation ayant été la cible d'un correctif n'est désormais plus détectée par RUSTInA.

Notons qu'un correctif est également généré dans le cas des potentielles fausses alarmes, bien que le cas est extrêmement rare en pratique – la qualité des corrections sera discutée en section 5.5.2.3.

Les tableaux 5.4 et 5.5 présentent l'impact des correctifs, soit la nouvelle répartition des fautes à l'échelle des projets et des blocs corrigés.

5.5.2.1 Résultats niveau projet

Corpus x86. Après correction automatique, RUSTInA confirme désormais que 178 projets ne contiennent aucun bloc violant la propriété de conformité, soit 61 de plus qu'avant la correction. Il reste toutefois 24 projets contenant des fautes du type « lecture d'un registre non déclaré ».

Corpus ARM. Après la correction automatique, les 3 projets étudiés sont désormais prouvés conformes par RUSTInA.

TABLE 5.4 – Nouvelle répartition niveau projet après correction par RUSTInA

	x86			ARM		
Projets observés	**202**			3		
– conformes	178	*+61*	88%	3	*+2*	100%
– avec problèmes bénins	0	*-31*	0%	0	–	0%
– avec violations dangereuses	24	*-30*	12%	0	*-2*	0%

5.5.2.2 Résultats niveau bloc

Corpus x86. Après la correction automatique, 1276 nouveaux blocs sont maintenant vérifiés conformes. Il reste tout de même 88 blocs lisant des registres sans les avoir initialisés (ce comportement sera discuté en section 5.6).

Corpus ARM. Après correction automatique, les 391 blocs sont désormais prouvés conformes.

5.5.2.3 Discussion sur la qualité des correctifs

Au delà des préférences personnelles de codage, il est intéressant de chercher à comparer la qualité des correctifs générés automatiquement par rapport à ce que produirait un développeur ayant une bonne connaissance de l'assembleur embarqué.

Table 5.5 – Nouvelle répartition niveau bloc après correction par RUSTInA

	x86			ARM		
Blocs observés	**2656**			392		
✅ – conformes	2568	*+1276*	97%	391	*+80*	100%
🛡 – avec problèmes bénins	0	*-1070*	0%	0	–	0%
❌ – avec violations dangereuses	88	*-206*	3%	0	*-81*	0%

Correctifs sur les *clobbers*. Dans un premier temps, notons qu'un correctif ajoutant des contraintes de registres (tel que l'ajout du *clobber*) sur un bloc qui est, à l'heure actuelle, compilé correctement, a peu de chance de modifier le binaire produit. En effet, le code fonctionne correctement car le compilateur n'utilise pas ce registre. Rajouter ces contraintes allient ainsi le meilleur des deux mondes, en garantissant plus de sûreté pour le futur sans compromettre le présent. Nous pouvons ainsi affirmer qu'au moins 75% des correctifs générés sont optimaux.

Correctifs sur les jetons. Lorsqu'un jeton en lecture seul est écrasé, il peut y avoir deux façons de modifier l'interface :

— une solution générique, que nous adoptons, consiste à ajouter un nouveau jeton en écriture et à le lier avec le jeton écrasé ;

— des solutions dédiées dépendant des contraintes sur les jetons. En effet, s'il existe déjà un autre jeton en écriture seule, modifié après que le jeton en lecture ne soit lu, il est possible de les unifier (*matching constraint*), sauvant ainsi une allocation de registre.

La solution préférée dépendra grandement du contexte dans lequel le bloc est inséré et de la pression qu'il exerce sur l'allocation des registres. Le gain d'une solution sur l'autre semble toutefois très limité. Dans ce cas également, le correctif apparaît comme au plus près de l'optimal.

Correctifs sur la mémoire. Les accès mémoire sont en revanche un sujet sur lequel les développeurs aguerris portent une grande attention. Le mot clé `"memory"` a, en effet, une portée importante et, utilisé à tort, peut dégrader les performances. Il reste nécessaire d'utiliser `"memory"` lorsqu'il n'y a aucun moyen de borner statiquement les accès mémoire. Lorsque les accès sont localisés à des déplacements constants par rapport à la valeur de jetons, il est préférable de remplacer `"memory"` par de nouveaux jetons de type indirect, comme expliqué en section 5.4. Nous verrons en section 5.5.3 que cette situation se présente dans 20% des correctifs qui auraient nécessité l'ajout du mot clé `"memory"` et auraient donc été jugé de moindre qualité.

Correctifs d'unicité. L'évaluation des correctifs de problème d'unicité est assez difficile car nous ne connaissons qu'un seul exemple de bloc ayant été corrigé par

le passé pour ce type de problème. Le correctif généré automatiquement permet de corriger le problème facilement et rapidement. Toutefois, il reste parfois préférable de repenser le bloc en partie ou en intégralité pour éviter le problème « par conception ».

Au final, le fait que de nombreux correctifs aient été soumis et acceptés par les développeurs (section 5.5.4) est également un gage objectif de qualité de nos correctifs.

Conclusion RQ4 et RQ5

Le prototype est capable de générer un correctif efficace et proche de l'optimal dans la grande majorité des problèmes détectés (au moins 75%). Au total, RUSTInA est capable de corriger 92% des problèmes remontés, même si certains correctifs pourraient avoir une alternative jugée plus intéressante par un humain. Ainsi, seuls 8% des violations remontées ne peuvent être corriger automatiquement.

5.5.3 Raffinement d'interface (RQ6)

Cette section cherche à évaluer l'efficacité de la technique de raffinement de la contrainte mémoire présentée en section 5.4.

Pour cela, nous considérons l'ensemble des 171 blocs assembleur embarqué qui contiennent initialement le mot-clé `"memory"` pour un usage autre que celui d'empêcher les optimisations du compilateur, auquel s'ajoute l'ensemble des 101 blocs pour lesquels le correctif généré par RUSTInA ajoute le mot-clé `"memory"`, soit un total de 272 blocs représentant une cible potentielle de raffinement.

RUSTInA réussit à éliminer complètement l'usage du mot-clé `"memory"` pour 41 blocs, soit environ 15% des cibles potentielles. Ceci comprend 21 blocs dont le mot-clé `"memory"` a initialement été ajouté par le développeurs (soit 12% des 171 blocs originaux) et 20 blocs sur 101 pour lesquels il n'est plus nécessaire de rajouter le mot-clé `"memory"` avec un correctif, améliorant ainsi la qualité du correctif (section 5.5.2.3) dans environ 20% des cas.

5.5.4 Impact sur la communauté des développeurs (RQ7)

Afin de confronter nos correctifs au regard des développeurs et concrétiser les résultats de notre étude, nous avons sélectionné 8 projets (à savoir, ALSA, FFMPEG, haproxy, libatomic_ops, libtomcrypt, UDPCast, xfstt, x264) pour lesquels nous avons soumis le descriptif des problèmes détectés et le « *diff* » à appliquer pour les corriger.

Ces 8 projets sont les mêmes que ceux examinés manuellement en section 5.5.1.4 – ils ont été choisis car ils couvrent une grande partie des problèmes jugés sérieux de notre corpus et/ou présentent un caractère sensible.

Nous avons ainsi soumis un correctif pour **114** blocs pour un total de **538** violations. La tableau 5.6 montre le détail de nos soumissions. Les correctifs ont été très bien reçus et à l'heure de la rédaction, **38** correctifs ont été acceptés et appliqués, corrigeant **156** problèmes d'interface dans **7** projets – les développeurs de FFMPEG étant encore en phase de relecture des changements.

Table 5.6 – Submitted patches

Projet	Catégorie	Statut	# blocs corrigés	# violation corrigées	Commit
ALSA	Multimédia	Accepté	20	64/64	01d8a6e, 0fd7f0c
haproxy	Réseau	Accepté	1	1/1	09568fd
libatomic_ops	Multi-threading	Accepté	1	1/1	05812c2
libtomcrypt	Cryptographie	Accepté	2	2/2	cefff85
UDPCast	Réseau	Accepté	2	2/2	20200328
xfstt	X Serveur	Accepté	1	3/3	91c358e
x264	Multimédia	Accepté	11	83/83	69771
FFMPEG	Multimédia	Relecture	76	382/382[1]	

[1] Incluant 27 problèmes manuellement corrigés.

Nous avons interrogé les développeurs de ces projets sur leur perception des problèmes de conformité à l'interface : ils reconnaissent le risque même si les compilateurs actuels ne produisent pas toujours de bugs. Ils sont d'autant plus enclins à appliquer les correctifs lorsque ceux-ci ne modifient pas le résultat de la compilation actuelle. Ils ont également montré de l'intérêt à utiliser RustInA pour évaluer les régressions en cas de changement de blocs assembleur – ce qui est toutefois une opération supposée rare.

Processus de soumission et examen des correctifs. Il est important de noter que la soumission des correctifs prend un temps considérable et ce pour deux raisons :

— sur le plan technique, les correctifs sont générés vis-à-vis du code après que le préprocesseur C ait été appliqué. Bien que souvent trivial, il est nécessaire de porter les modifications sur le code original. Cette tâche peut devenir plus complexe en présence d'un grand nombre de *macros* ;

— les correctifs doivent être examinés par l'équipe de développement avant d'être intégrés. Il est nécessaire de se tenir disponible pour participer aux discussions et ainsi expliquer ou justifier les changements apportés. Il est également parfois nécessaire, pour répondre aux exigences du projet, de retravailler la « *merge request* » ou d'appliquer une batterie de tests sur nos changements.

Nous pensons cependant que cet investissement de temps est nécessaire pour obtenir un impact réel et rendre plus sûr le développement de logiciels usant d'assembleur embarqué.

5.6 Étude qualitative des problèmes de conformité

Cette section a pour objectif d'essayer de comprendre la présence et le maintien d'autant de problèmes d'interface dans la base réelle de code ainsi que leur impact réel. Les problèmes rencontrés sont observés sous le prisme historique, retraçant une partie des changements et des limitations majeurs des compilateurs. Enfin, les

progrès techniques des compilateurs nous laissent à penser que si rien ne change, les problèmes qui peuvent paraître bénins ou irréalistes aujourd'hui pourront devenir sérieux dans le futur.

Cette section sert ainsi à la fois de conclusion mais aussi de motivation à ce chapitre dédié à la conformité à l'interface. Nous abordons cette problématique à travers 3 questions de recherche :

RQ8. Existe-t-il des similitudes entre les problèmes détectés en section 5.5.1 ?

RQ9. Le cas échéant, quel sont les raisons qui « empêchent » ces problèmes de se transformer en bug ?

RQ10. Enfin, est-il possible de créer des bugs à partir des blocs fautifs ?

5.6.1 Motifs récurrents d'interfaces mal formées (RQ8)

Pour commencer, nous avons identifié 6 motifs (*pattern*), dénotés de P1 à P6, présentés dans le tableau 5.7 et responsables de 91% des problèmes rencontrés (1989/2183) ou encore 80% des problèmes jugés sensibles (792/986).

TABLE 5.7 – Erreurs d'interface récurrentes

Motif	Catégorie	Contexte additionnel	# observé	
P1	`"cc"`	–	1197	55%
P2	registre `%ebx`	compilation *Position Independent Code*	30	1%
P3	registre `%esp`	utilisation de `push` et `pop`	5	0%
P4	`"memory"`	fonction composée d'un unique bloc assembleur	285	13%
P5	registres MMX	fonction composée d'un unique bloc assembleur	363	17%
P6	registres XMM	les registres XMM représentent un état global	109	5%

Dans chaque cas, une information sur les entrées ou les sorties du bloc est omise et cela se concentre sur un petit nombre de registres architecturaux (`"cc"`, `%ebx`, `%esp` ou les extensions SIMD) ou les accès mémoire.

Nous pensons que les motifs P2 et P6 sont intentionnels là où les motifs P1, P3, P4 et P5 seraient plutôt dûs à une méconnaissance de l'assembleur embarqué ou à de mauvaises habitudes. Le motif P4 pourrait également être une tentative maladroite des développeurs pour gagner en performance.

Nous allons voir dans la section suivante ce que ces motifs ont en commun et tenter d'expliquer pourquoi, avec les compilateurs de l'époque jusqu'à aujourd'hui, ils ne produisent pas toujours des bugs observables.

5.6.2 « Protections » historiques implicites de ces motifs (RQ9)

Lors de l'écriture d'un bloc assembleur, l'oubli d'un *clobber* est assez fréquent. Ceci dit, si cet oubli provoque immédiatement un bug observable, il sera corrigé immédiatement. Les problèmes d'interface qui ont persisté au fil du temps et des commits sont ainsi ceux qui, à l'instar d'une partie des comportements indéfinis du

C, fonctionnent correctement un temps, jusqu'à ce que de subtils changements du contexte (code, option de compilation, etc.) ne modifient la donne.

Nous avons identifié 3 types de « protection » favorisant l'apparition des mauvaises pratiques P1 à P6 :

— les choix officieux d'un compilateur ;

— les limitations techniques d'un compilateur ;

— le contexte « maîtrisé » de compilation.

Le tableau 5.8 résume cette association de protections.

TABLE 5.8 – Contexte protecteur

Motif	Catégorie	Protection	Nature
P1	`"cc"`	`"cc"` est un *clobber* par défaut en x86	choix historique
P2	registre `%ebx`	`%ebx` n'est pas alloué lors de la compilation PIC	limitation
P3	registre `%esp`	`%esp` n'est pas alloué dans les blocs assembleur	choix historique
P4	`"memory"`	absence de traitement interprocédurale	limitation
P5	registres MMX	absence de traitement interprocédurale (les registres MMX sont sauvegardé par l'appelant)	limitation (ABI)
P6	registres XMM	génération d'instructions XMM désactivée	configuration

Choix d'un compilateur et comportement « historique ». Ainsi, le grand nombre d'omission du mot clé `"cc"` vient du fait qu'il est notoirement connu que GCC (et par mimétisme Clang) considèrent le registre de condition comme systématiquement écrasé par un bloc assembleur x86 [2]. L'oubli n'étant pas « sanctionné », l'utilisation du mot clé `"cc"` n'est que rarement utilisé et avant tout à titre de « documentation ».

L'utilisation de `push` et de `pop` dans le bloc assembleur (P3) est passé inaperçu car le compilateur utilisait exclusivement le registre `%ebp` sauf si l'option `-fomit-frame-pointer` était donnée explicitement au compilateur (comportement qui a changé comme nous allons le voir en section 5.6.3).

Nous jugeons dangereux de fonder le bon fonctionnement du programme sur des comportements non documentés du compilateur qui sont susceptibles de changer d'une version à une autre sans aucun avertissement.

Limitations techniques des compilateurs. Les limitations techniques des compilateurs participent aussi à l'apprentissage et la diffusion de mauvaises pratiques. Le compilateur GCC (avant la version 5) était techniquement incapable d'allouer le registre `%ebx` quand le code était compilé en mode PIC (*Position Independant Code*), poussant les développeurs à l'utiliser sans le déclarer. Tout se passait bien tant que la valeur de `%ebx` était restaurée manuellement.

2. En x86, la grande majorité des instructions mettent à jour le registre de conditions.

L'absence d'optimisations interprocédurales poussées a pu faire croire au bon fonctionnement de blocs « cachés » derrière un appel de fonction. En effet, le bloc profite alors implicitement du traitement conservateur lié au respect de la convention d'appel, à la fois sur la mémoire, mais aussi sur les registres qui doivent être sauvegardés par l'appelant ou par l'appelé, bloquant des optimisations indésirables.

Ce type de protection est évidement jugé fragile car les compilateurs ne cessent de faire des progrès pour déployer des optimisations toujours plus puissantes. Les techniques avancées « d'*inlining* » rendues possible grâce au « *Link Time Optimisation* » (`-lto`) sont de bon exemples de progrès pouvant mettre à mal ces protections.

Contexte de compilation maîtrisé. Finalement, la protection apparaissant la plus légitime est celle consistant à paramétrer le contexte de compilation pour empêcher l'utilisation d'optimisations problématiques – par exemple en désactivant la génération de certains type d'instructions.

Nous jugeons ce type de protection comme insuffisant car les projets évoluent et avec eux, les besoins de compilation. En outre, les personnes en charge de ces projets peuvent changer. Les nouveaux responsables peuvent alors être moins au fait des choix et options garantissant le bon fonctionnement de l'assembleur embarqué. Enfin, les blocs assembleur embarqué peuvent simplement être « copier-coller » entre projets sans que les options ne soient mises à jour en conséquence.

Afin de valider nos soupçons sur la fragilité de ces protections, nous allons mettre à épreuve ces protections dans des scénarios pouvant favoriser l'apparition de bugs.

5.6.3 Robustesse des « protections implicites » (RQ10)

Notre objectif est de montrer à l'aide de petites expérimentations que les compilateurs actuels, utilisent les failles de l'interface des blocs non conformes et peuvent ainsi produire des bugs. Les résultats de ces expérimentations sont résumés dans le tableau 5.9.

Table 5.9 – Résultat d'expérimentations

Motif	Catégorie	Robuste	Menaces	Bugs connus
P1	`"cc"`	✅	changement de politique de GCC[1]	–
P2	registre `%ebx`	❌	mise à jour du compilateur (GCC ≥ 5)	`64d81cd`
P3	registre `%esp`	❌	mise à jour du compilateur (GCC ≥ 4.6)	`30cea1b`
P4	`"memory"`	❌	« copié-collé », optimisations (*inlining*)	`cefff85`
P5	registres MMX	❌	« copié-collé », optimisations (*inlining*)	–
P6	registres XMM	❌	« copié-collé »	–

[1] Il y a eu des discussions dans la liste de diffusion de GCC pour changer ce comportement [Boe15].

Dans un premier temps, notons que nous avons retrouvé des traces de bugs s'étant déjà manifestés, sous la forme de commit, pour les motifs P2, P3 et P4. Nous allons maintenant détailler les résultats de nos expérimentations pour chacun des motifs.

Motif P1. Le motif P1, basé sur un choix de compilateur historique toujours en application, est resté robuste sur l'architecture x86. Si la présence du mot clé `"cc"` n'est pas « obligatoire », le rajouter reste toutefois le comportement le plus sûr vis à vis du futur. Nous avons ainsi retrouvé des discussions sur la liste de diffusion de GCC en faveur d'un changement de politique [Boe15]. Il reste nécessaire sur les architectures ne bénéficiant pas de traitement particulier telle que ARM.

Motif P2. GCC ne souffre plus de sa limitation sur `%ebx` depuis la version 5. Clang, quant à lui, n'a jamais eu de restriction similaire. Un bug (64d81cd) a d'ailleurs été découvert dans libatomic_ops lorsque Clang a été utilisé pour la première fois.

Motif P3. Les options par défaut de GCC ont changé lors du passage à la version 4.6 pour intégrer l'option `-fomit-frame-pointer`. Depuis lors, les petites fonctions utilisent le registre de pile `%esp` pour adresser les variables locales. Ce registre étant implicitement modifié par les instructions `push` et `pop`, les références de variables locales données en entrée d'un bloc assembleur peuvent être décalées avant d'être accédées, résultant en des problèmes fonctionnels (30cea1b).

Motif P4. Les optimisations des compilateurs devenant de plus en plus agressives avec le temps (par exemple, `-lto`), le motif P4 basé sur l'absence de raisonnement interprocédural est devenu fragile. En effet, le compilateur aura tendance à copier le corps des petites fonctions dans la fonction appelante (*inlining*) et les fonctions contenant uniquement de l'assembleur embarqué font partie des cibles favorisées. Une fois la barrière de la convention d'appel tombé, le compilateur croit à tort que le bloc n'accède pas à la mémoire, perturbant le calcul de dépendances de données. Le compilateur peut alors ignorer une partie des affectations ou propager la valeur d'une affectation précédente au-delà du bloc.

Motif P5. Pour les mêmes raisons que le motif P4, le motif P5 est supposé fragile. En effet, le compilateur, une fois la fonction « *inlinée* » ne saura pas que le bloc utilise les registres MMX. Cela peut poser problème si le code appelant utilise ces registres, ce qui peut se produire dans deux cas :

— une optimisation de « vectorisation » ;

— la présence de calculs en virgule flottante car les registres MMX sont physiquement partagés avec les registres flottants 387.

Dans les deux cas, les compilateurs préféreront utiliser les registres XMM plutôt que MMX. Cependant, les registres MMX restent une ressource matérielle non utilisée dont les compilateurs pourraient tirer partie. Il est d'ores et déjà possible d'utiliser

l'option `-fpmath=sse+387` pour exploiter les deux types de registres. Pour s'assurer de la réalité du problème, nous avons essayé de compiler 10 blocs présentant ce problème d'interface avec un main réalisant du calcul en virgule flottante. Nous avons utilisé l'option `-fpmath=387` afin de favoriser la présence de bug. Le résultat du calcul était faux dans chacun des cas.

Motif P6. Le motif P6 apparaît robuste à première vue. Pourtant, les pratiques de développement impliquent régulièrement de copier du code d'un projet à un autre. Dans ce cas, si les options de compilation ne sont pas les mêmes dans le projet cible, cette copie augmente les risques qu'un bug apparaisse. Une expérience similaire à la précédente nous permet de s'assurer que copier « hors contexte » 10 blocs provoque des bugs fonctionnels.

De plus, il est important de noter que les registres XMM sont partagés entre plusieurs extensions architecturales. Ainsi, désactiver une certaine extension, par exemple SSE2 avec l'option `-mno-sse2`, n'empêche pas forcément le compilateur de produire des instructions pour les extensions suivante, par exemple AVX.

Conclusion RQ8, RQ9 et RQ10

Nous avons identifié 6 motifs récurrents responsables de la grande majorité des problèmes rencontrés de conformité. Le bon fonctionnement de ces motifs repose sur des protections implicites comme des choix historiques des compilateurs, des faiblesses ou des limitations des optimisations du compilateur ou encore la maîtrise du contexte de compilation. Nous jugeons ces types de protection fragiles et nous avons démontrés à l'aide d'exemple que 5 de ces motifs peuvent effectivement provoquer des bugs avec les compilateurs actuels.

5.7 Discussion

5.7.1 Validité des expérimentations

Comme discuté en section 4.5, le code assembleur considéré est quantitativement et qualitativement représentatif de l'usage de l'assembleur embarqué. Il comprend *tous* les blocs assembleur embarqué x86 trouvés dans le code source de la distribution Debian *Jessie* ainsi qu'un ensemble de blocs de l'architecture ARM.

Notre prototype est basé sur la plateforme BINSEC qui a déjà été utilisée dans des études de cas notables [BDM17, DBF+16, DBR20, DBR21, FMB+16] et a été intensément testée [KFJ+17].

Les résultats obtenus ont été manuellement vérifiés sur une centaine de blocs et les retours des développeurs montrent que RUSTInA soulève des problèmes qu'il faut corriger – une partie des correctifs proposés est déjà acceptée.

5.7.2 Limitations

La vérification de l'interface ne peut se faire que si la sémantique des instructions est connue. De ce fait RUSTInA souffre du manque de support des instructions de « flottants » et d'opérations système, non gérées dans BINSEC (et rarement gérées par les plateformes d'analyse de binaire en général).

Instruction flottante. Ici, un support minimal limité aux informations sur les entrées/sorties permettrait déjà d'obtenir des premiers résultats intéressants. Si ces informations sont une sur-approximation correcte, RUSTInA sera capable de garantir la conformité à l'interface. En revanche, cela pourrait éventuellement introduire des faux positifs.

Système. La formalisation proposée considère le bloc assembleur embarqué comme un calcul déterministe pour convertir des entrées bien identifiées de l'environnement C en sorties. Les instructions système lisent souvent des registres matériel cachés ou lisent et écrivent dans des emplacements mémoire inconnus du C. De telles instructions apparaissent alors comme non-déterministe et violeront la propriété de respect du cadre ou de l'unicité. Il serait possible d'étendre la définition de la conformité mais cela n'aurait de sens que si la syntaxe GNU est mise à niveau pour que l'interface puisse faire référence à des registres système. En effet, l'assembleur embarqué ne se prête pas, en l'état, à ce type d'instructions.

Jeux d'instruction. Comme discuté en section 4.5.2, notre prototype supporte actuellement les jeux d'instructions décodés par la plateforme BINSEC, à savoir x86-32 et ARMv7. Ce n'est pas une limitation conceptuelle puisque notre approche fonctionne sur la représentation intermédiaire et devrait supporter les nouvelles architectures ajoutées à BINSEC sans effort supplémentaire – des expérimentations sur l'architecture x86-64bit sont en cours.

5.7.3 Syntaxe Microsoft

Comme expliqué dans la section 3.2.1, l'assembleur embarqué proposé par VisualStudio (*inline MASM*) ne souffre pas des mêmes problèmes d'interface que la version GNU. Chaque instruction assembleur est une primitive connue du compilateur : il en connaît sa sémantique, ses effets de bord et ne requiert pas l'écriture d'une interface par le développeur. En contrepartie, cette syntaxe ne supporte qu'un sous-ensemble du jeu d'instructions x86-32.

5.8 Comparaison à l'état de l'art

Conformité à l'interface. Fehnker et al. [FHRS08] abordent le sujet de la conformité de l'interface de bloc assembleur embarqué ARM mais seulement vis à vis de la vérification – la génération de correctifs ou de raffinements ne sont pas examinés.

Leur définition de conformité restreint la détection aux seuls problèmes de non-respect du cadre (les problèmes d'unicité ne sont pas évoqués) et se base sur des observations syntaxiques plutôt que sémantiques, réduisant grandement la précision et la fiabilité de leur analyse. Leur technique requiert un analyseur syntaxique pour chaque architecture supportée. Bien que non conceptuelle, cette limitation limite grandement son applicabilité. Leur implémentation est ainsi fortement liée à l'architecture ARM dont le jeu d'instructions est plus simple que la famille x86.

Lien avec la traduction de code assembleur. Corteggiani et al. [CCF18] ont proposé une traduction automatique de l'assembleur (incluant l'assembleur embarqué) vers la représentation intermédiaire de LLVM à des fins de vérification (cet aspect sera discuté en section 6.7). Il n'est pas fait mention d'une quelconque tentative de vérification de l'interface. Il serait envisageable de vérifier certaines des propriétés de conformité de l'interface à la sortie de ce type de traduction. Toutefois, cela pose les problèmes suivants :

— dans la mesure où le bloc d'assembleur embarqué est concrétisé en assembleur avant la traduction, des informations essentielles sont perdues et empêchent la vérification des propriétés d'unicité, qui n'est plus exprimable au niveau C ou LLVM ;

— si l'outil réalise des optimisations de code lors de la traduction (élimination de « code mort », propagation ou ré-ordonnancement, mise en forme SSA ou renommage, etc.), il est possible que des informations sur l'affectation de registres soient perdues.

En outre, la traduction de code assembleur correcte et dans l'optique de vérifier ce code, est bien plus gourmande en ressources que nos techniques et la vérification de la conformité a de la valeur par elle-même. De ce fait, traduction et vérification de la conformité sont deux approches complémentaires qui méritent d'être traitées à différents niveaux.

5.9 Conclusion

Ce chapitre présente et formalise la propriété de conformité à l'interface. Pour garantir qu'un bloc puisse être compilé correctement quelque soit le choix du compilateur, son code et son interface doivent être cohérents :

framing-condition **:** le cadre doit être respecté et aucun effet de bord, lecture ou écriture, ne doit être omis ;

unicity-condition **:** l'allocation de registres et la substitution des jetons ne doit pas pouvoir impacter la sémantique du code.

Nous avons présenté, implémenté et évalué dans le prototype RUSTInA une méthode sûre de vérification automatique de la conformité à l'interface, basée sur des analyses flot de données. Nos résultats ont permis de mettre en lumière un nombre

important de blocs présentant une violation de la conformité (environ 11% du corpus d'exemple `x86`), et de proposer automatiquement des correctifs de qualité pour une large partie d'entre eux (92%).

Si jusqu'à présent les problèmes de conformité ne s'expriment pas souvent sous la forme de bugs observables, les menaces qui pèsent sur les codes utilisant des blocs fautifs sont à prendre au sérieux vis à vis des évolutions à venir des compilateurs. La possibilité de corriger automatiquement les problèmes de conformité offerte par RUSTInA devrait toutefois permettre d'éviter que des bugs ne se déclarent dans le futur. Au final, sept projets bénéficient déjà des correctifs proposés par RUSTInA.

Chapitre 6

TInA : portage vers le C

Sommaire

Nous nous attaquons maintenant aux problèmes engendrés par la vérification de la combinaison entre code C et assembleur embarqué. Comme nous avons pu le voir en section 2.3, les outils d'analyse actuels ne supportent pas bien cette extension des compilateurs :

— soit ils l'ignorent, rendant l'analyse incorrecte ;

— soit ils s'arrêtent, rendant l'analyse incomplète ;

— soit ils font des sur-approximations grossières, rendant l'analyse imprécise, au point de devenir inexploitable.

Ces raisons freinent l'adoption des outils de vérification pour du code contenant de l'assembleur embarqué.

Pour répondre aux problèmes posés, nous écartons la piste consistant à développer un nouvel analyseur dédié au code mixte C et assembleur. Cette option est en

effet trop coûteuse en termes de développement et l'outil serait certainement en retard sur toutes les analyses de niveau C. Aussi, nous préférons l'approche consistant à traduire le code assembleur en un code C sémantiquement équivalent, c'est-à-dire qui aura les même effets observables sur l'environnement d'exécution. Cette approche nous permet ainsi de réutiliser tels quels les outils existants d'analyse de niveau C.

Il existe une multitude de traductions envisageables mais toutes n'offrent pas le même degré d'intelligibilité pour les outils. Nous voulons que le code C produit ressemble suffisamment à du code C « naturel » pour des analyseurs automatiques. L'objectif est d'obtenir des résultats d'analyse qui puissent être exploitables.

Cette approche a déjà été exploré par Maus et al. [MMS08] pour analyser du code assembleur x86 sur Windows mais leur technique ne cible pas la syntaxe GNU, ne fonctionne que pour l'architecture x86 32bit et produit un code C de bas niveau qui n'est pas adapté à tous les outils d'analyse de code C. Corteggiani et al. [CCF18] proposent de traduire le code assembleur dans la représentation intermédiaire de LLVM. Si leur technique est plus générique et pourrait fonctionner sur différentes architectures, le code LLVM produit est également de très bas niveau. En outre, ces deux travaux proposent chacun une traduction *ad hoc* sans garantie d'équivalence entre l'assembleur original et le code généré.

Dans ce chapitre, nous proposons notre propre technique de traduction de notre représentation intermédiaire vers le C, présentée en section 6.1. Cette traduction est générique car elle repose sur l'extraction de la sémantique de l'assembleur embarqué (section 4.4). Contrairement aux approches antérieures, notre traduction ne se limite pas à une transformation systématique de la syntaxe, produisant un code bas niveau et difficile à analyser (section 6.2). En effet, afin de faciliter la vérification du code C produit par des outils existants, nous proposons un ensemble de passes de simplification qui éliminent une partie des constructions de bas niveau et retrouvent des abstractions de plus haut niveau (section 6.3). Enfin, pour garantir la fiabilité de notre approche, chaque bloc assembleur traduit est validé par une passe de validation dédiée, qui s'assure que le code C généré est observationnellement équivalent au code original.

6.1 Architecture générale de l'approche

Le processus de traduction prend place après l'extraction de la sémantique. La figure 6.1 montre la vue d'ensemble du processus.

La représentation intermédiaire originale est conservée en tant que référence (*Referential IR*) et elle servira à valider que le processus n'a pas modifié le comportement du programme.

Traduction. Plusieurs passes de simplification (*Transformations*) sont appliquées sur le programme pour retrouver des structures de plus haut niveau. Ces passes seront détaillées dans les sections 6.3.1 à 6.3.5

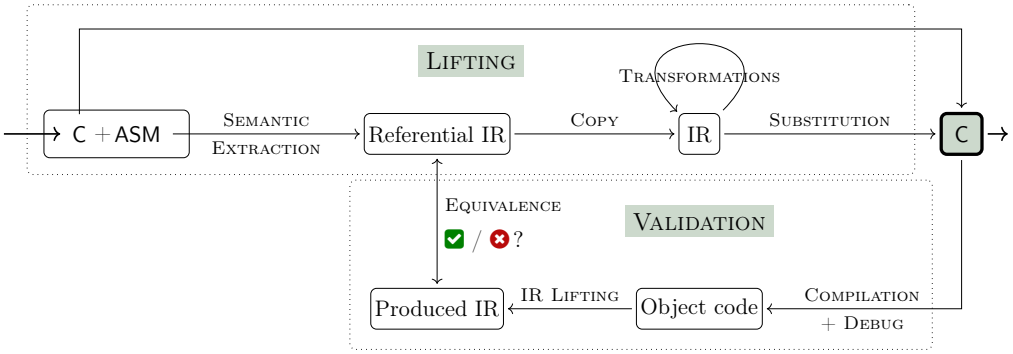


Figure 6.1 – Vue générale de TInA

Une fois le code simplifié, il est traduit dans la syntaxe du C. Cette étape peut être réalisée indépendamment des simplifications mais donnera alors un résultat plus difficile à analyser. Le code C obtenu est ensuite réinséré à la place du bloc assembleur dans le fichier source (*Substitution*).

Validation. Le code C traduit est à destination d'outils de vérification formelle. Les transformations de code n'étant pas vérifiées, nous passons ensuite par une étape de validation de la traduction. Cela consiste à comparer au niveau de la représentation intermédiaire de l'assembleur le code de référence à celui de sa traduction. Le code est compilé dans un environnement maîtrisé pour faciliter son désassemblage (*Compilation + Debug*).

Si les codes passent le test d'équivalence (*Equivalence*), le code C est prêt à être utilisé pour la vérification.

6.2 Première traduction

Afin de passer le blocage purement syntaxique lié à la présence de code assembleur dans du code C, il est possible de transcoder les opérations réalisées en assembleur dans la syntaxe du C. Cette opération est facilitée par la proximité des deux langages – C et représentation intermédiaire.

Notre représentation intermédiaire, définie en section 4.2.1, est donnée en rappel dans la figure 6.2.

Ainsi, on notera que :

— les types C ont des tailles directement corrélées avec la taille des registres de la machine hôte, et donc des entités manipulées par l'assembleur ;

— les expressions C et assembleur partagent la majorité des opérateurs – seuls les opérateurs manipulant la taille des opérandes n'ont pas d'équivalents directs en C ;

— les (micro-)instructions de l'assembleur sont entièrement exprimables en C.

L'approche directe et que nous qualifions de naïve consiste donc à proposer une traduction systématique (au niveau octet) de la représentation intermédiaire vers le

$$
\begin{array}{rcl}
\texttt{asm}^{\iota} & := & \text{begin} : \mu\texttt{label} \,;\, \text{body} : \mu\texttt{label} \to \mu\texttt{instr} \,;\, \text{end} : \mu\texttt{label} \\
\mu\texttt{instr} & := & \mu\texttt{oper}\ \text{goto}\ \mu\texttt{label} \\
\mu\texttt{oper} & := & \texttt{lvalue} \leftarrow \texttt{expr} \,;\, | \ \text{if}\ \texttt{expr}\ \text{then goto}\ \mu\texttt{label} \,;\, | \\
\texttt{lvalue} & := & \texttt{variable} \mid \texttt{@[}\ \texttt{expr}\ \texttt{]} \\
\texttt{expr} & := & \texttt{bitvector} \mid \texttt{lvalue} \mid \texttt{unop expr} \mid \texttt{binop expr expr} \\
\texttt{unop} & := & \neg \mid - \mid \text{zext}_n \mid \text{sext}_n \mid \text{extract}_{j..i} \\
\texttt{binop} & := & \texttt{arith} \mid \texttt{bitwise} \mid \texttt{cmp} \mid \text{concat} \\
\texttt{arith} & := & + \mid - \mid \times \mid \text{udiv} \mid \text{urem} \mid \text{sdiv} \mid \text{srem} \\
\texttt{bitwise} & := & \wedge \mid \vee \mid \oplus \mid \text{shl} \mid \text{shr} \mid \text{sar} \\
\texttt{cmp} & := & = \mid <_u \mid <_s
\end{array}
$$

FIGURE 6.2 – Représentation Intermédiaire de l'assembleur (Rappel)

C. Les outils de vérification existants sont cependant inefficaces sur le code C ainsi généré (section 6.5.2). Nous avons identifié 3 menaces principales qui perturbent les analyseurs et les empêchent de fonctionner à leur plein potentiel sur un tel code bas niveau :

Données de bas niveau : les conditions ne sont pas naturelles car elles sont calculées à partir des indicateurs architecturaux (*flags*) – dont la retenue (*carry*) ou le débordement (*overflow*). De plus, les variables ne sont pas typées et la mémoire est vue comme une suite d'octets ;

Variables implicites : le code combine des entités logiques distinctes au sein d'un même registre ou d'une même zone mémoire en utilisant des opérations bit-à-bit (masquage) ;

Structures implicites de boucle : les éléments qui dépendent du compteur de boucle de haut niveau sont séparés en de multiples calculs de bas niveau où le lien logique entre les éléments est « perdu ».

Nous présentons dans la suite cette première traduction des éléments de la syntaxe de notre représentation intermédiaire vers la syntaxe du C. Nous présenterons ensuite (Section 6.3) des passes de simplification destinées à retrouver un code de plus haut niveau.

Déclarations. La première étape consiste à déclarer chacun des registres machine en une variable globale ayant un type entier « non signé » (`unsigned`) correspondant à sa taille en bits. L'état du processeur est ainsi simulé dans une machine « virtuelle » décrite en C.

Contrôle de flots. Pour chacune des instructions, il est nécessaire de poser une étiquette C (notée `L`*label*`:`) en amont – elle servira de cible pour les sauts `goto` produit en C. La traduction du bloc commence par l'instruction `goto Lbegin;` et se termine par l'étiquette `Lend: ;`.

μinstructions. L'affectation ← est traduite directement en affectation C « = ». Le saut inconditionnel goto et le saut conditionnel if then goto sont directement traduit par leur équivalent C.

Emplacements. Les emplacements `lvalue` sont soit des variables globales ayant été déclarées dans la phase de **déclaration**, soit des accès mémoire traduits par l'opérateur de déréférencement « ∗ ».

Expressions. Les constantes bit vecteur sont traduites par des entiers de valeur équivalente. Les emplacements, variables et accès mémoire sont traduites comme décrit ci-avant. Les opérateurs communs entre le C et l'assembleur sont utilisés tels quels. Les opérateurs sect_n et concat peuvent être exprimés à partir des autres opérateurs comme décrit dans le paragraphe **sucre syntaxique** de la section 4.2.1. Pour rappel, les correspondances sont :

— concat a b = $((\mathsf{zext}_{\mathsf{sizeof(b)}}\ \mathtt{a})\ \mathsf{shl}\ \mathsf{sizeof}(\mathtt{b}))\ \mathsf{v}\ (\mathsf{zext}_{\mathsf{sizeof(a)}}\ \mathtt{b})$;

— sext_n a = $(-(\mathsf{zext}_{n-\mathsf{sizeof(a)}}\ \mathtt{a}_{\{\mathsf{sizeof(a)}-1\}}))$ :: a.

L'opérateur $\mathsf{extract}_{j..i}$ peut être décomposé en une opération de décalage à droite et une opération de masquage : $\mathsf{extract}_{j..i}$ a = $(\mathtt{a}\ \mathsf{shr}\ i)\ \wedge\ ((1\ \mathsf{shl}\ (j - i + 1)) - 1)$. L'opérateur zext_n est simplement traduit en *cast* C vers un type de taille égale ou supérieure à n bits.

Expression plus large que 64 bits. Les types standard du C ne permettent pas de représenter des bit vecteurs de plus de 64 bits (`long long int`). *Nous n'avons pas rencontré de tels expressions dans nos expérimentations.* Toutefois, un encodage des opérations, similaire à ce qui est fait dans la librairie de calcul arithmétique multiprécision GMP, reste envisageable.

6.3 Simplifications au service de la vérification

L'objectif de la traduction est d'obtenir un code C qui préserve la sémantique du bloc original mais également avec de bonnes propriétés de « vérifiabilité ». Pour cela, le code généré doit ressembler à du code C suffisamment haut niveau sur lequel les outils de vérification ont l'habitude de travailler.

Nous allons maintenant détailler nos 5 passes de transformation :

1. la propagation de types provenant de l'interface (section 6.3.1) ;
2. la reconstruction de prédicats de haut niveau (section 6.3.2) ;
3. la séparation des variables groupées (section 6.3.3) ;
4. la propagation d'expressions symboliques (section 6.3.4) ;
5. la normalisation de compteurs de boucle (section 6.3.5).

6.3.1 Propagation de types

Les opérations sur les vecteurs de bits et la mémoire sont directement exprimables sur des entiers non signés (`unsigned int`) et des pointeurs non typés (`void *`). Toutefois, l'absence de type peut rendre la compréhension du programme plus difficile sur deux points :

— la conversion entre types signés et non signés peut entraîner une perte de précision sur des abstractions telles que les intervalles ;

— l'arithmétique native des pointeurs et la conversion de pointeurs ne sont pas correctement supportées par tous les outils niveau C.

Les deux sont évitables en propageant les types C des entrées à travers la représentation de l'assembleur. Le type de chaque sous-expression est calculé en suivant les règles de typage du C. Des conversions peuvent être introduites pour respecter les contraintes de signe associées à certains opérateurs assembleur.

En assembleur, un même registre peut être réutilisé plusieurs fois pour stocker des informations de types différents. Pour éviter toute ambiguïté, le code est mis sous forme pseudo-SSA [CFR+91]. Cette forme permet de déclarer chaque instance de registre comme une variable locale indépendante dont le type est correctement déterminé par l'expression qu'elle reçoit.

6.3.2 Reconstruction de prédicats de haut niveau

La majorité des architectures mettent à jour des indicateurs (*flags*) de conditions lors de l'exécution, utilisés ensuite par les conditions de branchement. Par exemple, sur les architecture x86 et ARM, les indicateurs les plus utilisés sont :

— z – indicateur de valeur à 0 ;

— s – indicateur de signe ;

— c – indicateur de retenue (*carry*) ;

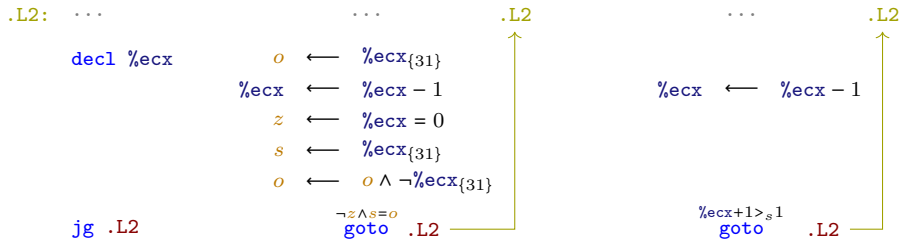
— o – indicateur de débordement (*overflow*).

Le calcul de ces indicateurs peut complexifier le programme au point de le rendre difficilement compréhensible par des outils non spécialisés. En effet, la façon dont les indicateurs sont mis à jour dépend du type d'opération (arithmétique, bit-à-bit, etc.), ce qui signifie que la même instruction de saut conditionnel peut avoir plusieurs significations.

La figure 6.3a montre l'exemple de calcul d'une condition de fin de boucle. Le compteur `%ecx` et décrémenté et les indicateurs z, s et o sont mis à jour. L'instruction `jg` (*jump if greater*) se traduit alors par le test $\neg z \wedge s = o$. Nous utilisons la technique proposée par Djoudi et al. [DBG16] pour retrouver un prédicat de haut niveau, considéré comme plus « naturel », comme ici, $\%\texttt{ecx} + 1 >_s 1$ [1] (figure 6.3b). Pour trouver les prédicats adaptés, nous utilisons une

1. Il pourrait être tentant de simplifier un degré de plus en $\%\texttt{ecx} >_s 0$. La condition ne serait toutefois plus strictement équivalente dans le cas où `%ecx` vaudrait `INT_MAX` à cause des débordements signé non défini en C.

technique est basée sur l'équivalence prouvée par un solveur SMT entre la condition de saut et un petit ensemble de prédicats (=, $<_s$, $>_s$, etc.) et opère une substitution d'expression en cas de succès [2].



(a) Avant reconstruction (b) Après reconstruction

FIGURE 6.3 – Comparatif de la reconstruction de prédicat de haut niveau

6.3.3 Séparation de variables groupées (*unpacking*)

L'assembleur embarqué contient souvent des optimisations exploitant la capacité maximale des registres en groupant plusieurs valeurs (*packing*). Il est par exemple possible de lire en mémoire 4 caractères (8 bits) simultanément pour les ranger dans un registre de 32 bits et y appliquer des masques à base d'opérations bit à bit avant de les renvoyer en mémoire. Une classe entière de registres (*Single Instruction Multiple Data*) dédiée à ces opérations groupées est ainsi présente sur les architectures communes.

Ces opérations se traduisent dans la représentation intermédiaire de niveau assembleur en de multiples extractions et concaténations, ramenés en C à un ensemble de masques et de décalages qui mettent à mal les abstractions des analyseurs.

Notre approche est une heuristique qui consiste à retrouver et séparer les entités indépendantes qui sont groupées dans un même conteneur. Le principe est le suivant : si une sous-partie d'une variable est lue ou écrite dans le code (comme par exemple $\texttt{\%eax}_{\{7..0\}}$) alors cette sous-partie a de forte chance de représenter une variable indépendante. Aussi, nous créons une nouvelle variable correspondante et la substituons à chaque référence de la sous-partie de la variable originale. Cette opération répétée jusqu'à ce que toutes les extractions sur des variables aient disparu. Les variables originales qui ne sont plus référencées seront éliminées.

Pour éviter d'avoir à itérer jusqu'à trouver un point fixe où toutes les extractions ont été remplacées, nous procédons à un remplacement systématique en 3 étapes :

1. une passe en avant éclate chaque affectation de variable de taille 8×2^k en différents recouvrements (sous-variables) dont la taille est de la forme 8×2^i avec $i \leq k$. Ainsi, une affectation à `%eax` sera subdivisée en affectations indépendantes pour {{`al`, `ah`, `eax_16_23`, `eax_24_31`}, {`ax`, `eax_16_31`}, {`eax`}}.

2. Les indicateurs qui ne sont plus utilisés seront supprimés par une passe d'élimination de code mort.

Cette opération fait grandir la taille du code ;

2. dans un même temps, chaque opération d'extraction est remplacée par la (sous-)variable correspondante (taille et position) ;
3. une passe d'élimination de code mort élimine les sous-variables non utilisées. Si des sous-parties superposées peuvent cohabiter (par exemple, `al`, `ax` et `eax`), nous avons constaté que, dans la majorité des cas, seul un des variants persistait à la fin de l'élimination de code mort. La taille du code n'augmente donc que très rarement à la fin du processus.

La figure 6.4 présente un bloc assembleur qui charge 2 caractères (`char`) dans dans un registre de 16 bits (modificateur `k`) avant d'en faire l'addition en adressant la partie basse (modificateur `b`) et la partie haute (modificateur `h`). Sans contrainte supplémentaire, le compilateur choisira le registre `%ax` et ses sous-parties `%al` et `%ah`.

```
extern const
unsigned char src[2];
unsigned char sum;
__asm__
("movzxw   %1,  %k0\n\t"
 "addb     %h0, %b0\n\t"
 : "=&Q" (sum)
 : "m" (src));
```

FIGURE 6.4 – Bloc assembleur avant traduction

La figure 6.5 montre le résultat de la traduction littérale. La traduction utilise des masques de bits rendant le code plus complexe que nécessaire.

```
unsigned int __tina_eax0;
unsigned int __tina_eax1;
__TINA_BEGIN_0__: ;
__tina_eax0 = (*src) | (*(src + 1) << 8);
__tina_eax1 = (0xffffff00 & __tina_eax0) | (0xff &
  ((0xff & (__tina_eax0 >> 8)) + (0xff & __tina_eax0)));
sum = 0xff & __tina_eax1;
__TINA_END_0__: ;
```

FIGURE 6.5 – Résultat de la traduction littérale

En revanche, une fois les variables correctement séparées, la traduction donnée en figure 6.6 représente plus clairement la fonction originale du bloc.

6.3.4 Propagation d'expressions symboliques

Cette passe s'inspire des techniques d'optimisation des compilateurs pour les adapter à nos besoins particuliers. En effet, la taille des problèmes autorise des sim-

```
unsigned char __tina_ah;
unsigned char __tina_al0;
unsigned char __tina_al1;
__TINA_BEGIN_0__: ;
__tina_al0 = *src;
__tina_ah = *(src + 1);
__tina_al1 =
  __tina_ah + __tina_al0;
sum = __tina_al1;
__TINA_END_0__: ;
```

FIGURE 6.6 – Résultat de la traduction simplifiée

plifications très agressives, et ce, en prenant en compte les spécificités des opérateurs de bas niveau.

Notre approche combine une propagation systématique d'expressions avec un contrôle à posteriori pour annuler les propagations qui ne participent pas à la simplification de code.

La propagation systématique d'expressions repose sur l'idée d'offrir davantage d'opportunités aux simplifications syntaxiques de rencontrer des motifs à réécrire. Le principal inconvénient de cette propagation inconditionnelle apparaît à l'émission du code où la même expression peut être dupliquée un grand nombre de fois. La taille des expressions peut alors devenir excessive et avec elle la taille du code généré. Aussi, nous évitons l'explosion de la taille du code en annulant (*revert*) à posteriori les propagations qui n'ont pas participé à la simplification du code par les règles de réécritures. Les expressions qui sont retrouvées inchangées sont ainsi remplacées par la variable originale.

Cette passe se déroule en 5 étapes :

1. en premier lieu, une analyse de flots de données collecte l'ensemble des paires composées d'une variable et de son expression symbolique en chaque point du programme (Propagation d'expressions, page 68) ;
2. nous propageons inconditionnellement les expressions symbolique en remplaçant les occurrences des variables ;
3. les règles de réécriture sont appliquées sur les expressions afin de tenter d'en réduire la taille ou la complexité ;
4. la relation entre variables et expressions symboliques est inversée et les expressions qui n'ont pas été simplifiées par les règles de réécriture sont remplacées par la variable originale ;
5. enfin, une passe d'élimination de code mort supprime les variables qui ne sont plus utilisées.

Concernant les règles de réécriture, nous utilisons à la fois des règles standard, telles que l'associativité et la commutativité de certains opérateurs arithmétiques, et des règles dédiées pour les aspects spécifiques aux vecteurs de bits. L'ensemble

complet de règles est donné en annexe B. Voici quelques exemples de règles spécifiques :

— arithmétique en complément à 2 :

$$x_{\{\texttt{sizeof}(x)-1\}} \hookrightarrow x <_s 0 \tag{6.1}$$
$$\neg x + 1 \hookrightarrow -x \tag{6.2}$$
$$x \oplus -1 \hookrightarrow \neg x \tag{6.3}$$

— extractions :

$$(\mathsf{sext}_n x)_{\{j..i\}} \overset{\texttt{sizeof}(x)\leq i}{\hookrightarrow} x_{\{\texttt{sizeof}(x)-1\}} \,?\, -1 : 0 \tag{6.4}$$

— distributivité :

$$(c \,?\, x : y) \oplus z \hookrightarrow c \,?\, x \oplus z : y \oplus z \tag{6.5}$$
$$(c \,?\, x : y) - (c \,?\, a : b) \hookrightarrow c \,?\, x - a : y - b \tag{6.6}$$

La figure 6.7 montre un bloc assembleur réalisant le calcul de la valeur absolue de son argument `%eax`. Bien que connue et utilisée par les compilateurs, cette forme optimisée n'est pas évidente à reconnaître.

```
cltd            # sign extend eax in edx
xor %edx, %eax  # 1-complement eax if eax < 0
sub %edx, %eax  # add one to eax if eax < 0
```

FIGURE 6.7 – Optimisation de la fonction valeur absolue sans branchement

Sa représentation intermédiaire originale (sous forme SSA) est donnée dans la figure 6.8 où elle est simplifiée à la volée en utilisant les règles précédentes, résultant en une expression claire et reconnaissable : $\mathsf{eax}_2 \leftarrow \mathsf{eax}_0 <_s 0 \,?\, -\mathsf{eax}_0 : \mathsf{eax}_0$.

$$\begin{aligned}
&\mathsf{tmp}_{64} \leftarrow \mathsf{sext}_{64}\ \mathsf{eax}_0 \\
&\mathsf{edx}_0 \leftarrow \mathsf{extract}_{32..63}\ \mathsf{tmp}_{64} \overset{(6.4)+(6.1)}{\hookrightarrow} \mathsf{edx}_0 \leftarrow \mathsf{eax}_0 <_s 0 \,?\, \texttt{0xffffffff} : 0 \\
&\mathsf{eax}_1 \leftarrow \mathsf{eax}_0 \oplus \mathsf{edx}_0 \overset{(6.5)+(6.3)}{\hookrightarrow} \mathsf{eax}_1 \leftarrow \mathsf{eax}_0 <_s 0 \,?\, \neg\mathsf{eax}_0 : \mathsf{eax}_0 \\
&\mathsf{eax}_2 \leftarrow \mathsf{eax}_1 - \mathsf{edx}_0 \overset{(6.6)+(6.2)}{\hookrightarrow} \mathsf{eax}_2 \leftarrow \mathsf{eax}_0 <_s 0 \,?\, \text{-}\mathsf{eax}_0 : \mathsf{eax}_0
\end{aligned}$$

FIGURE 6.8 – Résultat des simplifications syntaxiques

6.3.5 Normalisation de compteurs de boucle

Cette passe cherche à mettre en évidence le lien entre l'indice de boucle (compteur d'itérations) et la valeur des autres variables, en particulier les relations affines de la forme $a \times x + b$ où x est l'indice de boucle.

En effet, bien que les deux formes suivantes soient strictement équivalentes, nous avons constaté que certains outils d'analyse étaient sensibles à la « forme

syntaxique » et préféraient : `for (int i = 0; i < N; i++) T[i] = C;` plutôt : `for (char *t = T; t < T + N; t++) *t = C;`. Or, pour des raisons d'économie de registres, le code bas niveau aura tendance à favoriser la seconde forme.

Notre approche consiste donc à retrouver le lien implicite entre l'indice de boucle et la valeur des variables à l'intérieur de cette boucle et de factoriser le code pour rendre ce lien explicite. Cette transformation s'apparente au calcul du domaine de Gauge [Ven12] en interprétation abstraite.

Nous nous concentrons donc sur les variables qui s'auto-incrémentent (respectivement s'auto-décrémentent) à l'intérieur de la boucle. Ces variables sont caractérisées par une initialisation de la forme `v = I` et un auto-incrément de la forme `v = v + k` où `k` est une expression constante dans tout le corps de la boucle.

La forme du motif étant syntaxique, il est préférable d'appliquer cette passe après la propagation d'expressions qui a pour effet de normaliser le code.

Nous réalisons la transformation en (au plus) 3 étapes :

1. **changement de base** : remplace la valeur initiale `I` par 0 et chaque occurrence de la variable `v` (à l'exception de l'auto-incrément) par `I + v` ;
2. **mise à l'échelle** : remplace la valeur de l'incrément `k` par 1 et chaque occurrence de la variable `v` (à l'exception encore de l'auto-incrément) par `k * v` ;
3. **factorisation** : unifie toutes les variables (qui sont désormais sous la forme `v = v + 1`) en une seule.

Regardons ces 3 étapes sur l'exemple de la figure 2.5. La figure 6.9 montre le code à la sortie de la propagation d'expressions. Les variables `edi` et `ecx` sont candidates à la normalisation.

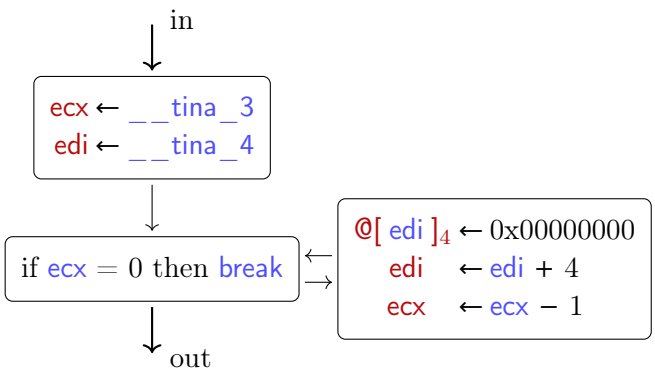


FIGURE 6.9 – Bloc à simplifier

Les initialisations `__tina_3` et `__tina_4` sont remplacées par des 0 et sont réinsérées aux points d'occurrence d'ecx et d'edi dans la figure 6.10. Dans la figure 6.11, les deux incréments 4 et −1 sont normalisés pour devenir 1. En même temps, les occurrences de d'edi et d'ecx sont multipliées par ces facteurs. Finalement, les variables d'ecx et d'edi sont unifiées pour ne garder que la variable ecx (figure 6.12).

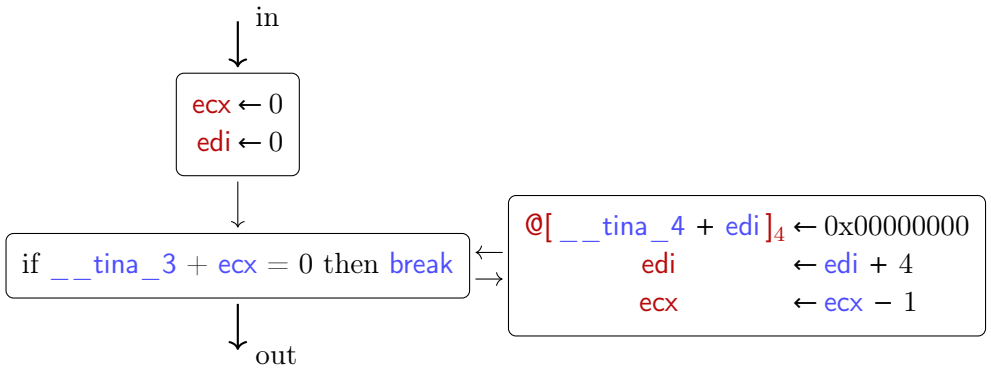


FIGURE 6.10 – Étape 1 : changement de base (*rebasing*)

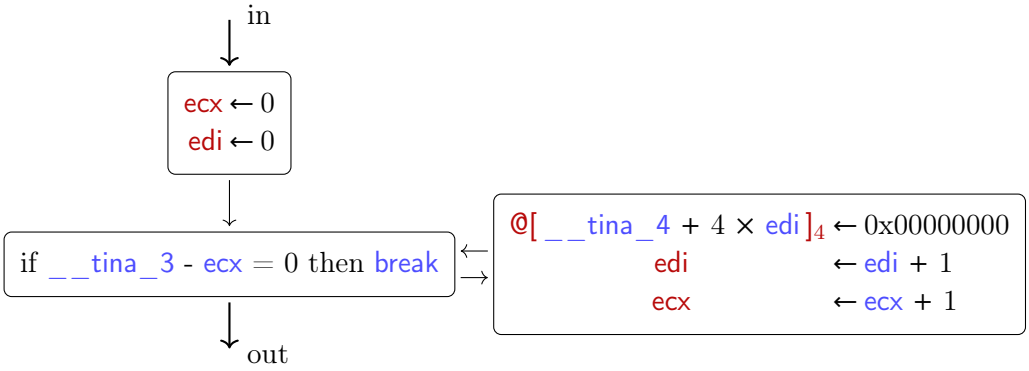


FIGURE 6.11 – Étape 2 : mise à l'échelle (*rescaling*)

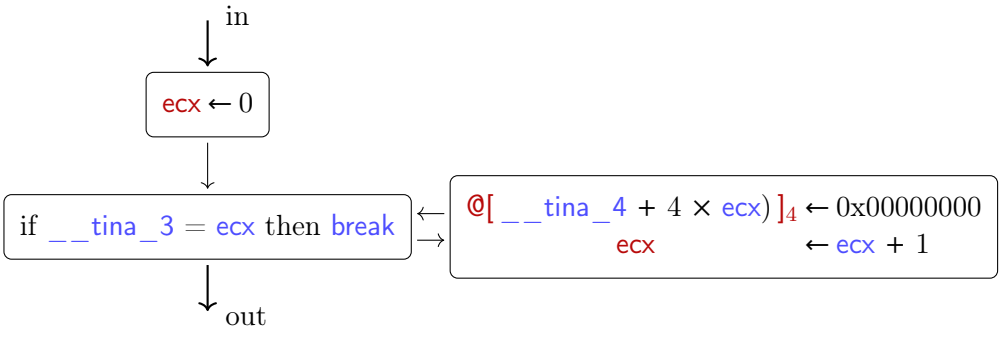


FIGURE 6.12 – Étape 3 : après factorisation

6.4 Validation automatique

Dans la mesure où ni la traduction vers le C (bien que très directe), ni les passes de simplification ne sont prouvées correctes, notre approche propose une validation a posteriori du code généré afin d'augmenter la confiance accordée à l'outil.

L'approche générale consiste dans un premier temps à tester l'équivalence de code entre la représentation intermédiaire originale de l'assembleur et la représentation intermédiaire de l'assembleur généré par compilation de la traduction C. En cas d'échec de la preuve d'équivalence, il est possible de recourir à des techniques classiques de test comme le test d'entrées aléatoires (*fuzzing* [MHH+19]) – ceci dit, nous verrons en section 6.5.1 qu'il n'a jamais été nécessaire d'en arriver là, le test d'équivalence présenté dans cette section se montrant très efficace.

L'équivalence de programmes est un problème difficile dans le cadre général, mais nous exploitons ici les similitudes entre les deux codes pour arriver à nos fins – puisqu'un des codes est *par construction* directement dérivé du second. Notre technique est conçue de manière à garantir que la traduction générée possède le *même graphe de flot de contrôle* que le code original. Avec cette contrainte forte, si tous les blocs de base de la traduction et de l'original sont deux à deux équivalents, nous pouvons en conclure que les deux programmes sont équivalents. Notre processus de validation automatique se déroule ainsi en deux étapes : (1) l'appariement de blocs de base, conditionnée par l'isomorphisme des graphes de flot de contrôle (section 6.4.1) ; et (2) la vérification de l'équivalence des paires de blocs appariés (section 6.4.2).

Nous allons maintenant détailler ces deux étapes.

6.4.1 Isomorphisme de graphe de flot de contrôle

La première condition pour que la validation puisse avoir lieu est que chacun des blocs de base du code de la traduction soit associé à un bloc de base du code original.

Le problème d'isomorphisme de graphe est compliqué (NP-complet) dans le cas général mais se résout trivialement dans le cadre de TInA, ceci pour deux raisons :

— les passes de simplification sont conçues pour conserver le graphe de flot de contrôle. Le code C obtenu a donc exactement le même graphe ;

— le code est compilé sans optimisation (`-O0`). Les compilateurs produisent ainsi le code assembleur sans tenter de faire de transformations comme le déroulage de boucle (*loop unrolling*) qui pourraient détruire l'isomorphisme de graphe.

De plus, nous rajoutons des étiquettes au niveau du source et nous utilisons les informations de *debug* DWARF pour faciliter l'association des blocs de base entre les deux graphes de flot de contrôle.

Si les informations de *debug* venaient à manquer, une approche basée sur de l'énumération exhaustive (*brute force*) pourrait également obtenir de très bons résultats. Les noeuds (blocs de base) n'ont, en effet, qu'au plus 2 successeurs et les graphes rencontrés sont de très petite taille (un maximum de 21 noeuds). En outre, cette étape ne concerne que 8% et 21% des blocs assembleur x86 et ARM, le reste n'ayant qu'un seul bloc de base, l'appairage est trivial.

La figure 6.13 illustre l'association de blocs de base pour l'exemple de la figure 2.5 présenté dans le chapitre d'introduction.

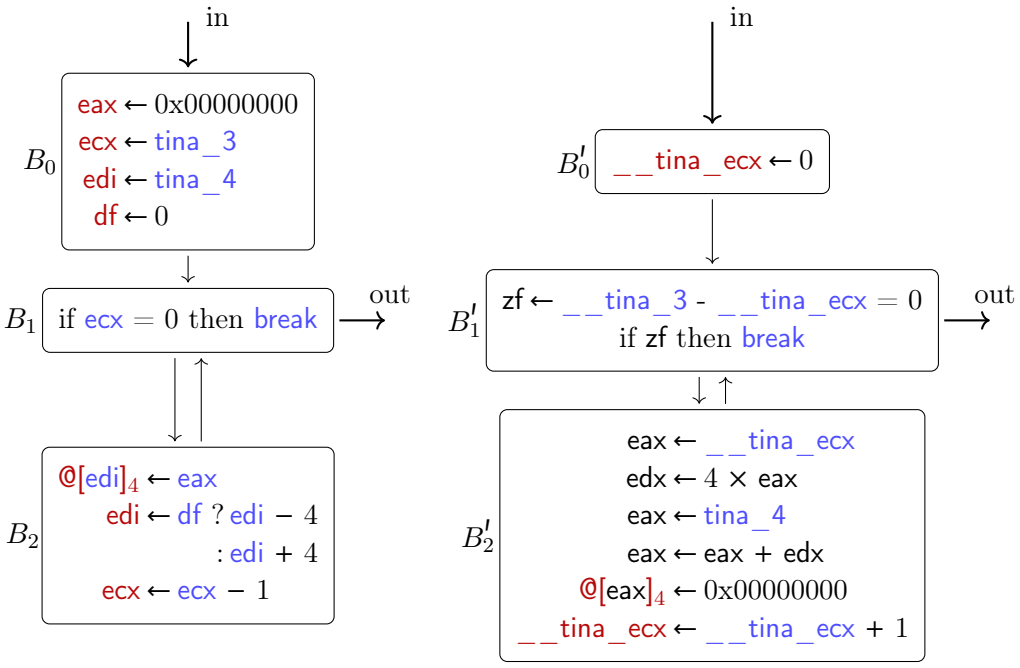


FIGURE 6.13 – Association de blocs de base

6.4.2 Équivalence de blocs de base

Un test d'équivalence est réalisé sur chacune des paires de blocs de base pour déterminer s'ils ont la même fonctionnalité. Un bloc de base peut-être vu comme une fonction prenant des entrées et renvoyant des sorties. Le test d'équivalence cherche à montrer que, pour les deux blocs de base associés, les sorties sont toujours identiques quand les entrées sont identiques.

Cela demande d'identifier les entrées et les sorties des deux blocs de base et de pouvoir les associer. L'un étant le résultat de la traduction de l'autre, l'association entre variables est une information propagée par le processus de traduction.

Les passes de simplification peuvent impacter le nombre de variables d'entrée et de sortie par rapport au programme original. Pour pouvoir établir les correspondances, les passes de simplification sont conçues pour enregistrer des méta-données

que nous décrivons ci-dessous pour chacune des passes.

Reconstruction de prédicats de haut niveau (**section 6.3.2**). Les méta-données permettent de retrouver le calcul original des indicateurs (*flag*) à partir des opérandes identifiés ;

Séparation de variables groupées (**section 6.3.3**). Les méta-données permettent de retrouver la position relative des sous-variables extraites par rapport à la variable originale ;

Propagation d'expression symbolique (**section 6.3.4**). Les méta-données permettent de retrouver la valeur des variables propagées entre les différents blocs ;

Normalisation des compteurs de boucles (**section 6.3.5**). Les méta-données enregistrent les équations linéaires découvertes entre les différentes variables d'une boucle.

Ces méta-données sont alors utilisées pour reconstruire les variables d'entrée et de sortie du bloc original qui auraient disparues.

La preuve d'équivalence est accomplie par un solveur SMT après que les deux blocs aient été transformée en formules et que les liens manquants soient comblés à l'aide des méta-données fournies par les passes de simplification. La requête demandée au solveur SMT est la suivante : « sachant que les entrées sont équivalentes, peut-il exister une différence dans les sorties ? ». Si la formule SMT est déclarée insatisfiable (*UNSAT*), les deux blocs implémentent exactement la même fonction et sont donc équivalents.

Exemple. La figure 6.14 donne la formule à traiter pour prouver l'équivalence entre les blocs B_2 et B_2' de la figure 6.13. Les deux blocs sont syntaxiquement différents à cause des simplifications et de la recompilation. De plus, les registres ne sont pas utilisés de la même façon :

— les registres du bloc original ont été traduits en variable dont le nom permet d'en retrouver l'origine (par exemple, `%ecx` et `__tina_ecx`) ;

— les registres du bloc généré sont utilisés dans des étapes de calculs intermédiaires.

À cause des simplifications, le nombre d'entrées et de sorties est différent. En effet, les passes de simplification ont déterminé que les entrées `%eax` et `df` sont respectivement égales aux constantes `0x00000000` et `false`, et qu'il existe une relation linéaire entre `%ecx` et `%edi` (`%edi = tina_4 + 4 × (tina_3 − %ecx)`). Ces méta-données nous permettent de faire le lien dans la formule logique.

La formule est complétée par l'assertion qu'au moins une des paires de sortie présente une différence. Le moteur de résolution ne trouvant aucun solution à cette formule (*UNSAT*), l'équivalence des deux blocs est prouvée.

; séquence d'instructions de B_2

$$\begin{aligned}
\mathsf{memory}_{\mathrm{out}} &= \mathrm{store}_4\ \mathsf{memory}_{\mathrm{in}}\ \mathsf{edi}_{\mathrm{in}}\ \mathsf{eax}_{\mathrm{in}} \\
\mathsf{edi}_{\mathrm{out}} &= \mathrm{if}\ \mathsf{df}_{\mathrm{in}}\ \mathrm{then}\ \mathsf{edi}_{\mathrm{in}} - 4\ \mathrm{else}\ \mathsf{edi}_{\mathrm{in}} + 4 \\
\mathsf{ecx}_{\mathrm{out}} &= \mathsf{ecx}_{\mathrm{in}} - 1
\end{aligned}$$

; séquence d'instructions de B_2'

$$\begin{aligned}
\mathsf{eax'}_0 &= \mathsf{__tina_ecx}_{\mathrm{in}} \\
\mathsf{edx'}_0 &= 4 \times \mathsf{eax'}_0 \\
\mathsf{eax'}_1 &= \mathsf{tina_4}_{\mathrm{in}} \\
\mathsf{eax'}_2 &= \mathsf{eax'}_1 + \mathsf{edx'}_0 \\
\mathsf{memory'}_{\mathrm{out}} &= \mathrm{store}_4\ \mathsf{memory}_{\mathrm{in}}\ \mathsf{eax'}_2\ 0\mathrm{x}00000000 \\
\mathsf{__tina_ecx}_{\mathrm{out}} &= \mathsf{__tina_ecx}_{\mathrm{in}} + 1
\end{aligned}$$

; liant à partir des méta-données d'entrée

$$\begin{aligned}
\mathsf{eax}_{\mathrm{in}} &= 0\mathrm{x}00000000 \\
\mathsf{df}_{\mathrm{in}} &= \texttt{false} \\
\mathsf{edi}_{\mathrm{in}} &= \mathsf{tina_4}_{\mathrm{in}} + 4 \times \mathsf{__tina_ecx}_{\mathrm{in}} \\
\mathsf{ecx}_{\mathrm{in}} &= \mathsf{tina_3}_{\mathrm{in}} - \mathsf{__tina_ecx}_{\mathrm{in}}
\end{aligned}$$

; liant à partir des méta-données de sortie

$$\begin{aligned}
\mathsf{edi'}_{\mathrm{out}} &= \mathsf{tina_4}_{\mathrm{in}} + 4 \times \mathsf{__tina_ecx}_{\mathrm{out}} \\
\mathsf{ecx'}_{\mathrm{out}} &= \mathsf{tina_3}_{\mathrm{in}} - \mathsf{__tina_ecx}_{\mathrm{out}}
\end{aligned}$$

; au moins une paire de sorties différente

$$\mathsf{memory}_{\mathrm{out}} \neq \mathsf{memory'}_{\mathrm{out}} \vee \mathsf{ecx}_{\mathrm{out}} \neq \mathsf{ecx'}_{\mathrm{out}} \vee \mathsf{edi}_{\mathrm{out}} \neq \mathsf{edi'}_{\mathrm{out}}$$

FIGURE 6.14 – Vérification de l'équivalence des blocs B_2 and B_2'

6.5 Evaluation expérimentale de TInA

Les étapes de simplification, de traduction et de validation ont été implémentées dans notre prototype TInA. Dans sa version finale, le code des algorithmes de simplification, de traduction vers la représentation intermédiaire de Frama-C et de validation de cette traduction ajoute environ 2kLOC supplémentaires à la base de TInA et de Frama-C. L'évaluation de TInA est faite à travers 4 questions de recherche :

RQ11. Combien de blocs supportés par TInA sont portés vers le C ?

RQ12. Combien de traductions sont automatiquement validées ?

RQ13. Comment se comportent les outils de vérification niveau C sur le code obtenu ?

RQ14. Quel est l'impact de chacune des optimisations sur les résultats des analyses niveau C ?

6.5.1 Traduction et validation (RQ11, RQ12)

Traduction. Notre approche de traduction, qu'elle soit basique (section 6.2) ou avec simplification (section 6.3) est systématique. Ainsi tous les blocs conformes à l'interface (après correction si nécessaire) sont traduits en code C, c'est à dire 2568 blocs x86 et 391 blocs ARM.

Validation. Sur ce corpus, la validation automatique fonctionne parfaitement, avec un taux de succès de 100% de traduction validée.

Performance. La traduction couplée à la validation prend en moyenne 0.7s par bloc. La faible complexité du code, de petite taille et avec un nombre de blocs de base réduit (voir tableaux 4.1 et 4.1b) permettent des passes de simplification agressives sans problème de passage à l'échelle.

Les résultats de la traduction et de la validation sont compilés dans le tableau 6.1.

6.5.2 Adéquation aux outils de vérification formelle (RQ13, RQ14)

Corpus d'évaluation. Nous avons sélectionné trois outils représentant les techniques formelles populaires utilisées dans l'industrie :

— KLEE [CDE08] pour l'exécution symbolique ;

— Frama-C EVA [Büh17] pour l'interprétation abstraite ;

— Frama-C WP [BBCD18] pour la vérification déductive.

Pour rappel, ces trois techniques ont été introduites dans la section 3.1.1.

Nous utilisons KLEE et Frama-C EVA pour détecter les erreurs pouvant se produire lors de l'exécution (*RunTime Error*). KLEE permet de trouver des RTEs là ou Frama-C EVA essaie d'en prouver l'absence. Dans les deux cas, l'analyse se lance à partir d'un contexte spécifiant les parties de l'environnement considérées comme

Table 6.1 – Application de TInA sur l'assembleur embarqué de Debian

(a) Corpus x86

	Total	ALSA	FFMPEG	GMP	libyuv
Blocs assembleur	3107	25	103	237	4
Supportés et conformes	2568 85%	25 100%	91 88%	237 100%	1 25%
Traduits	2568 100%	25 100%	91 100%	237 100%	1 100%
Temps cumulé	155s	1s	68s	8s	$<$ 1s
Validés	2568 100%	25 100%	91 100%	237 100%	1 100%
Temps cumulé	1372s	16s	200s	104s	$<$ 1s

(b) Corpus ARM

	Total	ALSA	FFMPEG	GMP	libyuv
Blocs assembleur	394	0	85	308	1
Supportés et conformes	391 99%	–	83 98%	308 100%	1 100%
Traduits	391 100%	–	83 100%	308 100%	1 100%
Temps cumulé	5s	–	$<$ 1s	4s	$<$ 1s
Validés	391 100%	–	83 100%	308 100%	1 100%
Temps cumulé	235s	–	48	187	$<$ 1s

symboliques ou abstraites. La préparation de ce contexte est ainsi facilement réutilisable d'un outil à l'autre. Les deux outils seront donc évalués sur le même ensemble de fonctions.

Nous avons sélectionné 58 fonctions issues des quatre projets ALSA, FFMPEG, GMP et libyuv. Ces fonctions ont été choisies pour représenter un large panel des opérations réalisées en assembleur embarqué, de la petite optimisation de calcul aux opérations sur des tampons (*buffer*).

Nous utilisons l'outil Frama-C WP pour démontrer des propriétés fonctionnelles du code. Cette tâche étant moins automatisable que précédemment, l'évaluation de WP est limitée à 12 fonctions. En effet, pour chaque fonction, il est nécessaire d'exprimer la ou les propriétés que nous souhaitons démontrer et, en présence de boucles, il est nécessaire d'ajouter manuellement les invariants de boucles.

Pour chacun des trois outils, nous détaillerons les résultats obtenus pour différents niveaux de simplification de la traduction. Le tableau 6.2 donne l'équivalence entre le nom donné à un niveau de simplification et la liste des simplifications qui sont appliquées (section 6.3).

TABLE 6.2 – Niveaux d'optimisation

PASSE	$O0$	$O1$	$O2$	$O3$	$O4$	$\overline{O3}$	$\overline{O2}$	$\overline{O1}$
Élimination de code mort	✓	✓	✓	✓	✓	✓	✓	✓
Mise en forme *pseudo-SSA*	✓	✓	✓	✓	✓	✓	✓	✓
Propagation de type	✓	✓	✓	✓	✓	✓	✓	✓
Reconstruction de prédicats de haut niveau	✗	✓	✓	✓	✓	✓	✓	✗
Séparation de variables groupées	✗	✗	✓	✓	✓	✓	✗	✓
Propagation d'expression symbolique	✗	✗	✗	✓	✓	✗	✓	✓
Normalisation des compteurs de boucles	✗	✗	✗	✗	✓	✓	✓	✓

Les niveaux $O1$ à $O4$ permettent d'évaluer l'effet de l'ajout d'une optimisation par rapport aux précédentes. Les niveaux $\overline{O1}$ à $\overline{O3}$ permettent à l'inverse de mesurer la nécessité d'une optimisation. Les autres combinaisons sont moins essentielles à l'évaluation et ne sont pas détaillées. Notons que le niveau $\overline{O4}$ est équivalent à $O3$.

6.5.2.1 Impact de TInA sur KLEE

Dans son état actuel, le moteur d'exécution symbolique KLEE ne prend pas en charge l'assembleur embarqué. Travaillant au niveau de la représentation intermédiaire de LLVM, l'exploration s'arrête lorsqu'un bloc assembleur est rencontré. Par conséquent, KLEE n'est pas adapté pour traiter les fonctions de notre corpus.

Nous avons lancé KLEE sur chacune des fonctions durant *10 min* (*timeout*) et ce pour chacun des niveaux de simplification (tableau 6.2). Nous cherchons à mesurer la vitesse d'exploration de chemins de KLEE par rapport au niveau de simplification. Plusieurs métriques sont compilées dans le tableau 6.3.

Table 6.3 – Impact de la traduction sur l'exploration de KLEE

	Traduction								
	Sans	$O0$	$O1$	$O2$	$O3$	$O4$	$\overline{O3}$	$\overline{O2}$	$\overline{O1}$
Fonctions explorées sans blocage	3	58	58	58	58	58	58	58	58
Fonctions explorées à 100%	✗	25	25	25	25	25	25	25	25
Temps cumulé	–	115s	115s	110s	103s	105s	110s	105s	105s
Chemins explorés	1.4M	1.5M	1.8M	4.6M	6.6M	6.6M	4.6M	4.6M	5.4M
		+6%	+22%	+150%	+45%	~	-30%	-30%	-19%

Prérequis. La première métrique est la présence ou non de code « bloquant » l'exploration. Sans aucune traduction, seules 3 des 58 fonctions ne rencontrent pas de bloc assembleur provoquant l'arrêt [3]. Quelque soit le niveau de la traduction, dès lors que les morceaux d'assembleur sont retirés, KLEE ne rencontre plus d'obstacles bloquants.

Durée d'analyse pour un nombre fixé de chemins. Nous étudions ensuite un premier sous-ensemble de 25 fonctions que l'outil arrive à explorer exhaustivement avant la fin du temps imparti (ici, 10 min). Le nombre de chemins explorés étant le même pour les traductions, nous avons observé le temps que mettait l'outil pour terminer l'exploration. La propagation d'expressions ($O3$) apparaît ici comme la simplification apportant le plus grand gain de temps. Cependant ces exemples sont trop petits pour que la différence soit réellement significative.

Nombre de chemins pour une durée fixée d'analyse. Penchons nous désormais sur les 33 fonctions restantes. La métrique est alors inversée, le temps est constant (10 min) et nous observons la variation du nombre de chemins explorés. Nous pouvons constater dans un premier temps que permettre l'accès aux chemins bloqués par l'assembleur n'augmente pas significativement le nombre de chemins explorés. Les résultats pour les différents niveaux de traduction semblent indiquer que le code traduit de façon trop littéral ($O0$) est significativement plus difficile à analyser. La séparation de variables groupées et la propagation d'expressions symboliques apparaissent toutes deux comme très efficaces et complémentaires. Ces deux passes ($O2$ et $\overline{O2}$) permettent l'exploration d'environ 4.6M de chemins. Toutefois, ce n'est que lorsque les deux sont actives que KLEE atteint ses meilleurs performances ($O3$). En revanche, les boucles étant déroulées par la technique d'exécution symbolique, la normalisation des indices de compteur de boucle n'a aucun impact visible ici.

3. Après analyse du code LLVM généré (*bytecode*), il s'avère que le compilateur Clang a remplacé, probablement par « identification de motif », les blocs effectuant une simple rotation (`rol` ou `ror`) par des primitives du langage.

6.5.2.2 Impact de TInA sur Frama-C EVA

Nous cherchons à mesurer l'évolution de la précision des domaines abstraits en fonction du niveau de simplification. L'impact de la traduction des morceaux d'assembleur dans le langage C peut être observé à différents niveaux, soit directement en regardant la valeur abstraite d'une variable en un point de programme, soit indirectement en regardant le nombre de faux positifs. Le tableau 6.4 rassemble nos critères d'évaluation.

Table 6.4 – Impact de $O4$ sur la précision d'EVA

Fonctions avec	ALSA	FFMPEG	GMP	libyuv	Total
Valeur de retour (non `void`)	0	9	10	1	20
Meilleures valeurs de retour	–	9	1	1	11 55%
Alarmes dans le code C initial	2	8	16	1	27
Réduction d'alarmes dans le C	2	8	12	1	23 85%
Nouvelles alarmes mémoire	12	2	3	0	17 26%
Impact positif	14	17	13	1	45 77%

De manière directe, nous avons observé que 11 des 20 fonctions retournant une valeur numérique (55%) gagnent en précision sur le domaine abstrait de la valeur de retour.

De manière indirecte, nous avons observé une diminution du nombre d'alarmes présentes dans le code C d'origine. Cela signifie que l'outil a réussi à gagner en précision dans le calcul de valeurs intermédiaires et en a déduit que certaines conditions du C étaient toujours vraies ou fausses. Ainsi sur 27 fonctions possédant des alarmes dans le code original, 23 (85%) présentent une réduction du nombre d'alarmes remontées.

En combinant les deux métriques, sur 34 fonctions qui retournent une valeur numérique ou possèdent des alarmes, 31 (91%) fonctions présentent une amélioration entre la version originale et la traduction.

Il est également important de prendre en compte les 17 fonctions qui présentent de nouvelles alarmes mémoire dissimulées initialement par la syntaxe de l'assembleur – rendant donc l'analyse possiblement incorrecte.

Alarmes cachées. Contrairement à KLEE, le greffon EVA de Frama-C supporte la syntaxe de l'assembleur embarqué. Ce support est toutefois limité puisqu'il est restreint aux informations inférées de l'interface. Les variables déclarées en écriture sont transposées en annotations logiques `assign`, ce qui signifie que ces variables, et seulement ces variables, peuvent changer de valeur. Les accès mémoires arbitraires dans l'assembleur (`"memory"`) sont alors problématiques car Frama-C ne les traitent pas. Dans ce cas, il avertit l'utilisateur qu'il pose des hypothèses optimistes, à savoir qu'aucune valeur en mémoire n'est modifiée, et que ces hypothèses peuvent rendre l'analyse incorrecte. Précisons que si l'interface n'est pas conforme (notion présentée dans le chapitre 5), l'analyse peut également devenir incorrecte sans que l'utilisateur ne soit averti.

Au final, nous observons un effet positif de la traduction produite par TInA sur les résultats de l'interprétation abstraite d'EVA pour 45 des 58 fonctions (77%).

Dans une optique comparative, le tableau 6.5 permet d'évaluer l'impact des différents niveaux de simplification.

Table 6.5 – Comparatif des optimisations sur EVA

	Traduction								
# Fonctions avec	Sans	$O0$	$O1$	$O2$	$O3$	$O4$	$\overline{O3}$	$\overline{O2}$	$\overline{O1}$
absence d'alarmes	✗	12	12	14	14	**19**	14	14	14
alarmes sur la mémoire ASM	–	29	29	29	21	17	29	21	21
erreurs sur la mémoire ASM	✗	1	1	1	2	2	1	2	2
Nombre cumulé d'alarmes									
C émises	231	184	184	177	177	177	177	184	177
ASM émises	–	316	244	199	165	128	199	171	253

La première métrique est le nombre de fonctions pour lesquelles aucune alarme n'est remontée par EVA. Si 19 fonctions ne déclenchent aucune alarme en $O4$, cela signifie qu'il y a au moins 5 fausses alarmes dûes à des imprécisions pour $O2$, $O3$, $\overline{O1}$, $\overline{O2}$ et $\overline{O3}$ et 7 pour $O0$ et $O1$.

Le nombre d'alarmes émises peut-être un bon indicateur de la précision. En règle générale, plus l'outil sera précis et moins il émettra de fausse alarmes, ce qui se traduit par une diminution du nombre total d'alarmes émises (fausses et vraies). Si toutes les passes sont bénéfiques pour réduire le nombre d'alarmes, les passes les plus importantes sont la propagation d'expressions et la normalisation des indices de boucle. Cette passe aide particulièrement l'interprétation abstraite à raffiner efficacement ses invariants de boucle.

À l'inverse d'une fausse alarme, une alarme pour laquelle l'outil est certain que la condition est vraie deviendra une erreur. Ainsi, plus l'outil est précis et plus le nombre d'erreurs est important. Cela nous a permis de détecter que deux fonctions du projet FFMPEG accèdent à l'indice -1 du tampon donné en paramètre. Ce comportement est connu des développeurs et il s'agit du fonctionnement normal de la

fonction dans le contexte où elles sont appelées. Dans le contexte créé pour nos expérimentations en revanche, ces accès étaient bel et bien illégaux.

6.5.2.3 Impact de TInA sur Frama-C WP

Nous avons manuellement annoté 12 fonctions, dont le corps est composé d'assembleur embarqué, pour vérifier une ou plusieurs propriétés fonctionnelles. Ces fonctions sont données dans le tableau 6.6.

Table 6.6 – Description des fonctions évaluées

Fonction	# Inst.	# Inv.	Description	Origine
saturated_sub	2	0	Maximum entre 0 la soustraction d'entiers	FFMPEG, GMP
saturated_add	2	0	Minimum entre `MAX_UINT` et l'addition d'entiers	FFMPEG, GMP
log2	1	5	Plus grande puissance de 2 d'un entier	libgcrypt
mid_pred	7	0	Élément médian entre 3 entrées	FFMPEG
strcmpeq	9	6	Égalité de chaîne de caractères	ASM snippet
strnlen	16	6	Longueur de chaîne de caractères (bornée par n)	ASM snippet
memset	9	5	Initialisation de tableau	UDPCast
count	8	4	Nombre d'occurrences d'un élément dans un tableau	ACSL by example
max_element	10	7	Index du premier élément le plus grand	ACSL by example
cmp_array	10	6	Égalité entre tableaux (SIMD)	ACSL by example
sum_array	20	7	Somme des éléments d'un tableau (SIMD)	ACSL by example
SumSquareError	24	69	Somme des différences au carré (SIMD)	libyuv

Ce corpus contient 4 fonctions de « nature numérique », 2 fonctions travaillant sur des chaînes de caractères et 6 fonctions sur des tableaux. De plus, 9 fonctions contiennent au moins une boucle et nécessitent des invariants de boucle. Les mêmes invariants (aux noms de variables près) sont donnés pour tous les niveaux de simplification de la traduction.

Nous évaluons l'impact de la traduction TInA sur un critère unique de succès de la démonstration des propriétés fonctionnelles. Le tableau 6.7 illustre les résultats obtenus.

Sans surprise, en l'absence de traduction, aucune propriété fonctionnelle ne peut être démontrée. En effet, tous les effets des blocs assembleur sont considérés comme non déterministes.

Pour cet outil, chaque passe d'optimisation a de l'importance, à l'exception de la normalisation des indices de boucle. Cette dernière n'est pas nécessaire car les invariants de boucle donnés manuellement remplissent déjà ce rôle. Cette passe peut toutefois permettre d'écrire des invariants de boucle plus lisibles, ce qui peut avoir un impact sur l'adoption de la technique. À l'exception des deux premières fonctions, l'outil WP échoue à démontrer les propriétés si une seule des passes d'optimisation est désactivée ($\overline{O1}$ à $\overline{O3}$), montrant qu'en règle générale chacune des passes est nécessaire et que l'outil ne fonctionnera de façon optimale que si elles sont toutes activées.

Table 6.7 – Impact sur les preuves de WP

	Traduction								
Fonction	Sans	$O0$	$O1$	$O2$	$O3$	$O4$	$\overline{O3}$	$\overline{O2}$	$\overline{O1}$
saturated_sub	✗	✓	✓	✓	✓	✓	✓	✓	✓
saturated_add	✗	✗	✓	✓	✓	✓	✓	✓	✓
log2	✗	✗	✗	✗	✓	✓	✗	✓	✓
mid_pred	✗	✗	✓	✓	✓	✓	✓	✓	✗
strcmpeq	✗	✗	✗	✗	✓	✓	✗	✗	✓
strnlen	✗	✗	✗	✗	✓	✓	✗	✗	✗
memset	✗	✗	✗	✗	✓	✓	✗	✗	✓
count	✗	✗	✗	✗	✓	✓	✗	✓	✗
max_element	✗	✗	✓	✓	✓	✓	✓	✓	✗
cmp_array	✗	✗	✗	✗	✓	✓	✗	✗	✗
sum_array	✗	✗	✗	✗	✓	✓	✗	✗	✗
SumSquareError	✗	✗	✗	✗	✓	✓	✗	✗	✗

Conclusion RQ13 et RQ14

Nos expérimentations ont permis de montrer que donner une traduction C des blocs assembleur était une nécessité pour utiliser correctement les outils d'analyse. Elles montrent également que toutes les traductions n'ont pas le même impact sur le bon déroulement de l'analyse et qu'une simple traduction, comme celle proposée en section 6.2 n'offre pas beaucoup d'avantages.

Chaque outil réagit de façon différente à la qualité de traduction mais tous montrent un effet très bénéfique des passes d'optimisation de TInA :

— la vitesse d'exploration de chemins de KLEE augmente fortement (400%), permettant de couvrir plus vite une plus large portion de code ;

— la précision d'EVA augmente, permettant de fournir des rapports plus pertinent à l'utilisateur et de réduire le nombre de fausses alarmes ;

— la « complexité » des obligations de preuve de WP diminue, permettant la preuve de propriétés fonctionnelles jusqu'alors indémontrables.

6.6 Discussion

6.6.1 Validité des résultats

Nous considérons trois des techniques de vérification les plus populaires (exécution symbolique, interprétation abstraire et vérification déductive), représentant la diversité des analyses de code, tant au niveau des objectifs (détection de *bugs*, vérification de l'absence d'erreurs à l'exécution, preuve de propriétés fonctionnelles) que de la technologie sous-jacente (propagation de domaines, exploration de chemins, raisonnement en logique du premier ordre). Les outils sélectionnés, KLEE et

Frama-C, sont bien établis et ont déjà été utilisés avec succès sur des études de cas industriels. Nous estimons ainsi que nos expérimentations démontrent les propriétés de généricité et de vérifiabilité de la traduction offerte par TInA.

6.6.2 Limitations

Notre méthode de traduction hérite des limitations de la technique d'extraction de représentation intermédiaire de l'assembleur (voir section 4.5.2) : les opérations sur les nombres en virgule flottante (`float`) et les opérations systèmes.

Notons par ailleurs que ces deux classes d'opération posent des problèmes spécifiques dans le cas de TInA.

Flottants. Le support des « flottants » reste un défis de taille pour de nombreuses techniques d'analyses (analyseurs de C, solveurs SMT, etc.). Aussi, pour le moment et même avec un support adapté dans le décodeur, l'analyse de bloc assembleur contenant ce type d'instruction resterait hors de portée de la plupart des analyses.

Opérations système. Les opérations système sont d'autant plus difficiles à traiter que chaque outil de vérification peut avoir sa propre stratégie pour les supporter (annotations, assertions, *builtins*, etc.). Toutefois, sachant qu'une grande partie des appels systèmes en C passe par des appels à des fonctions de la librairie standard et que ces fonctions peuvent être supportées nativement par les outils d'analyse, il pourrait être intéressant de traduire les instructions assembleur dédiées aux appels système dans la fonction de la libc associée.

6.7 Comparaison à l'état de l'art

Bien que quelques travaux antérieurs aient abordé la vérification de code assembleur à partir d'une traduction en C, la question de l'adéquation avec différentes techniques d'analyse n'a jamais été étudiée. Dans la suite, nous allons recenser les travaux apparentés à notre approche.

Traduction et vérification d'assembleur. Maus [MMS08, Mau11] propose une méthode pour simuler le comportement des instructions assembleur dans un interpréteur écrit en C. Ce travail a été utilisé dans le projet *Verisoft* pour vérifier le code d'un hyperviseur composé de nombreux blocs assembleur embarqué MASM. Sa technique se combine avec VCC [CDH+09] (outil de vérification déductive) pour écrire et prouver des conditions sur l'état de cet interpréteur. Là où nous cherchons à obtenir une traduction ne conservant que les éléments essentiellement fonctionnels du bloc assembleur, le code C obtenu par la technique de Maus contient tous les détails bas niveau, incluant les registres machine et les indicateurs de conditions. Les travaux suivants de Schmaltz et Shadrin [SS12] réutilisent les travaux de Maus pour vérifier le respect de l'ABI (*Application Binary Interface*) pour des fonctions

écrites en assembleur MASM. Ces travaux peuvent fonctionner sur l'assembleur embarqué Microsoft, qui reste de plus haut niveau que l'assembleur embarqué GNU (section 3.2.1). En outre, TInA atteint un plus haut niveau de généricité avec le support de plusieurs architectures et son indépendance par rapport aux différents dialectes assembleur.

Les travaux contemporains de Corteggiani et al. [CCF18] utilisent également la traduction d'assembleur pour leurs besoins. Leur objectif est de pouvoir appliquer l'exécution symbolique sur le code obtenu. Nos expérimentations en section 6.5.2 montrent qu'une traduction permettant de lever la barrière syntaxique pour KLEE (par exemple $O0$) pourrait ne pas être suffisamment haut niveau pour être réutilisée pour d'autres outils. En outre, la technique de traduction n'est pas détaillée et la question de la correction n'est pas abordée.

Myreen et al. [MGS08] vérifient du code écrit en assembleur. La traduction qu'ils proposent au niveau logique n'est pas simplifiée et reste de très bas niveau (correspondant à notre niveau basique). Leur approche offre des garanties supplémentaire sur la traduction des instructions assembleur en représentation intermédiaire (*binary lifter*) grâce à une comparaison avec un modèle d'architecture très détaillé.

Décompilation. La décompilation s'attaque au défi de retrouver les sources originales ou ressemblantes à partir desquelles le code binaire a été obtenu. Cet objectif est très difficile dans le cadre général et requiert un travail important pour restaurer les informations qui sont perdues lors de la compilation. Aussi, malgré de récents progrès dans le domaine [BLSW13], la décompilation reste largement un défi ouvert. La décompilation vise dans un premier temps à faciliter la compréhension des programmes (*reverse engineering*). Dans cette utilisation, la correction de l'approche n'est pas la préoccupation principale. Il n'est ainsi pas toujours requis de produire un code C valide (pouvant être recompilé).

Les travaux de Brumley et al. [BJAS11] utilisent le test pour augmenter la confiance dans la traduction obtenue. Une telle approche dépend de la présence d'un jeu de tests conséquent pour le code sous analyse. Les techniques de test automatique (*fuzzing*) peuvent aider à découvrir des bugs mais rencontrent des difficultés à explorer profondément le code.

Schulte et al. [SRN+18] développent une approche où la correction est atteinte par construction : ils utilisent une méthode de recherche par mutation pour trouver un code source qui compile exactement, à l'octet près, vers le code binaire sous analyse. Cette technique peut fonctionner sur de petits exemples, mais rencontre des difficultés à converger pour des exemples plus gros. Notons que l'assembleur étant écrit à la main, il est possible qu'il n'existe aucune combinaison de code C et de compilateurs capable de donner le même résultat.

Analyse de code mixe. Morrisett et al. [MWCG98] ont proposé un langage assembleur typé (*Typed Assembly Language*) pour assurer l'intégrité de la mémoire et du contrôle de flots d'exécution. Patterson et al. [PPDA17] ont étendu le travaille

en mélangeant du code assembleur typé avec un langage fonctionnelle. Nous avons emprunté des éléments de leur technique pour propager les types entre le C et les instructions assembleur de bas niveau.

Validation de traduction et équivalence de code. Nous avons utilisé le principe de la validation de la traduction a posteriori [SMK13, Nec00, Riv04], une technique qui est utilisée également dans le compilateur CompCert pour la validation de l'allocation de registres [BRA10]. Nous obtenons des garanties à l'aide d'outils (ici les solveurs SMT) très largement utilisés et testés. Ces outils sont utilisés en « boite noire » et nous évitent de faire la preuve complète de la correction de la chaîne de traduction.

6.8 Conclusion

Ce chapitre a présenté notre approche de traduction facilitant la vérification de code C contenant de l'assembleur embarqué avec des outils existants représentants de différentes approches formelles.

Notre approche est la première à se concentrer sur la qualité de la traduction du point de vue de ces outils. Aussi, ayant identifié les principaux problèmes que peut poser une traduction « littérale » de la représentation intermédiaire bas niveau vers le C, nous proposons une série de passes de simplification permettant de retrouver des structures ou des abstractions de plus haut niveau.

Notre approche se veut fiable pour pouvoir être utilisée dans un environnement de vérification formelle. Pour ce faire, chaque traduction est validée automatiquement pour s'assurer que la sémantique du code original a bien été préservée.

Finalement, notre approche a été implémentée et testée sur un large ensemble de blocs assembleur. Notre prototype TInA obtient d'excellents résultats de traduction et de validation (100% des blocs assembleur supportés). Des expérimentations sur trois outils de vérification ont notamment montré l'impact très positif des simplifications opérées par TInA pour rendre le code vérifiable de manière efficace pour les outils KLEE et Frama-C.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Bilan

Nous considérons dans cette thèse les problèmes liés à la vérification de code « mixte » intégrant de l'assembleur embarqué écrit en syntaxe GNU au sein de code source C. L'assembleur, par sa nature bas niveau, est une source importante d'erreurs. Il demande donc une attention particulière. S'ajoute également la complexité de l'interface GNU, réputée pour être elle aussi source d'erreurs, ce qui peut entraîner des problèmes de mauvaise compilation.

Nous proposons des outils et méthodes d'analyse de programme pour aider à pallier ces problèmes. Nos travaux comptent trois contributions principales.

Obtenir la sémantique de l'assembleur. Nous introduisons tout d'abord une sémantique opérationnelle de l'assembleur embarqué, intermédiaire entre le C et le binaire, et enrichie par les informations de l'interface. Nous nous appuyons sur les techniques d'extraction de sémantique pour code binaire, que nous étendons au cas de l'assembleur embarqué pour tenir compte de la nature des jetons et du modèle mémoire du C, dont sont issus les pointeurs donnés en entrée. Les jetons sont identifiés à l'aide d'un processus itératif : les instructions sont assemblées et désassemblées pour plusieurs affectations de jetons soigneusement choisies pour que les différences identifient avec certitude la position des jetons.

Une méthode d'extraction de la sémantique proposée a été implémentée dans la plateforme d'analyse BINSEC et testée avec succès sur un large corpus représentatif. Elle offre un taux de réussite très élevé sur des exemples réels : **85**% sur 3107 blocs assembleur x86 et **99**% sur 394 blocs assembleur ARM.

Vérifier la conformité à l'interface. Nous proposons ensuite de vérifier que le comportement des instructions assembleur embarquées est conforme au comportement déclaré par l'interface. Notre approche repose sur une formalisation originale de la sémantique de l'interface et des problèmes de conformité qui vient remplacer

la définition incomplète donnée par Fehnker et al. [FHRS08]. Le **respect du cadre** (*framing condition*) et la **condition d'unicité** sont les deux propriétés nécessaires et suffisantes pour garantir cette conformité. Nous proposons un ensemble de techniques basées sur l'analyse de flot de données et applicables sur la représentation intermédiaire de l'assembleur pour vérifier ces deux propriétés ainsi que des optimisations dédiées pour limiter les faux positifs au maximum (aucun faux positif sur le corpus d'évaluation).

Appliquées au corpus de codes de la distribution Debian Jessie, ces techniques nous ont permis de détecter des problèmes sérieux dans **11**% des blocs assembleur x86. Notre prototype RUSTInA offre la possibilité de générer automatiquement un correctif pour **70**% d'entre eux. Nous avons commencé à soumettre ces correctifs aux développeurs et 7 projets les ont déjà intégrés dans leurs dépôts de code.

Porter l'assembleur vers du C vérifiable. Nous proposons enfin une traduction de l'assembleur embarqué vers le C adaptée à la vérification. Contrairement aux approches usuelles qui se contentent de traduire littéralement le comportement des instructions assembleur, notre approche repose sur des passes de simplification automatique pour transformer la représentation intermédiaire. Ces passes ont été soigneusement conçues pour éliminer les difficultés additionnelles à la vérification automatisée liées aux constructions de bas niveau utilisées dans l'assembleur. Il en résulte une traduction moins complexe et mieux intégrée dans son contexte. Par ailleurs, une passe de validation dédiée permet de s'assurer que la traduction obtenue conserve la même sémantique que le code original.

Nous avons testé notre traduction avec les trois outils de vérification KLEE, Frama-C EVA et Frama-C WP, représentants respectifs de l'exécution symbolique, de l'interprétation abstraite et de la vérification déductive, et nous avons observé une amélioration significative de leurs performances, à la fois par rapport à une utilisation sans support assembleur dédié mais aussi par rapport à une traduction plus basique. Ainsi, le moteur d'exécution symbolique est capable d'explorer davantage de chemins pour le même budget de temps ; la précision de l'interprétation abstraite est augmentée et pour finir, la vérification déductive parvient à prouver un plus grand nombre de propriétés fonctionnelles.

7.2 Perspectives

Nous exposons pour conclure trois pistes de développements autour des résultats obtenus durant ces travaux.

Extension du support. La première piste de développement porte sur le plan technique et consiste à lever certaines des limitations qui peuvent gêner l'adoption de TInA et RUSTInA sur certains types de codes. Ces limitations discutées en sections 4.5.2, 5.7.2 et 6.6.2 concernent les instructions en virgule flottante et les opérations système.

Les instructions de type flottant sont difficiles à traiter de manière complète mais un support limité aux seules relations d'entrée-sortie peut s'avérer suffisant pour vérifier la conformité de l'interface. La tâche d'ingénierie se restreint à la seule identification des instructions et nécessite donc moins d'investissement et de connaissance dans l'architecture que le support complet. Une approche mixte, sémantique approchée pour les flottants et précise pour le reste des instructions, a de fortes chances de fonctionner car les flottants forment généralement une classe de donnée « à part » qui n'interagit que peu avec le reste de l'environnement.

Le traitement des instructions système dépend lui du type d'analyse qui sera appliqué sur la traduction. Des analyses comme l'exécution symbolique ont besoin d'une sous-approximation (résultat concret d'une exécution par exemple) là où d'autres, comme l'interprétation abstraite, nécessitent des sur-approximations (intervalle de valeurs, etc.). Nous pouvons cependant remarquer qu'il existe une relation entre certaines instructions système et des fonctions de la librairie standard (*wrapper*). Ces fonctions sont parfois supportées nativement par les outils de vérification. Identifier et traduire les primitives assembleur de type système dans un équivalent à un appel de la librairie standard est un moyen générique et pratique de supporter une partie du code hors de portée de notre approche actuelle. Ce type de traduction ne pourrait cependant pas profiter de la validation automatique.

Généralisation du processus de traduction. Nous pouvons envisager d'étendre en partie notre approche à des fonctions écrites entièrement en assembleur, voir directement en binaire (« décompilation »). Cela n'est cependant pas immédiat car l'assembleur embarqué bénéficie de bonnes propriétés structurelles telles que l'absence de saut dynamique, une taille relativement restreinte et des informations contextuelles provenant de l'environnement C. Sans les restrictions imposées par la syntaxe de l'assembleur embarqué, les étapes de désassemblage et d'extraction du graphe de flot de contrôle requièrent une plus grande attention. Notamment les sauts dynamiques et les appels de fonctions sont désormais à considérer. La gestion de la mémoire, et en particulier l'allocation des variables et le typage dans la pile, seront des éléments à améliorer pour maintenir la « vérifiabilité » de la traduction. Toutefois, la convention d'appel (*ABI*) et la signature de la fonction peuvent jouer ici un rôle similaire à l'interface de l'assembleur embarqué et permettre l'injection des abstractions C dans le code bas niveau. Il est aussi possible de retrouver une partie des bonnes propriétés de l'assembleur embarqué dans de petites fonctions étrangères (`extern`), ce qui nous conforte dans l'idée que des résultats d'un niveau équivalent peuvent être obtenus.

Vers un nouvel assembleur embarqué. L'assembleur embarqué est né d'un besoin d'optimisation qui, s'il est devenu moins essentiel qu'auparavant, n'a pas complètement disparu. Plusieurs aspects de la syntaxe limitent l'écriture de l'interface et il serait intéressant d'offrir de nouvelles capacités, en particulier sur le traitement des entrées de type mémoire. Les contrats seraient plus faciles à écrire et

moins ambigus s'il était possible de faire explicitement la distinction entre entrées directes et indirectes et de déclarer des contraintes d'allocations plus fines comme le « *non-aliasing* » entre deux opérandes (plutôt que tous d'un coup). Enfin, la capacité de référer directement à des noms de registres systèmes ou architecturaux améliorerait à la fois la documentation du bloc et la compréhension du compilateur sur ces effets.

À défaut de changer la syntaxe établie, nous pourrions envisager de proposer une syntaxe alternative plus sûre pour les utilisations futures, à l'instar du langage Rust qui vient de se doter d'un nouvel assembleur embarqué [d'A20]. Outre le sucre syntaxique rendant l'écriture de bloc assembleur plus aisée, le changement porterait sur l'inversion du type d'hypothèse prise par le compilateur. Là où les modificateurs et mots clés servent actuellement à inhiber les optimisations indésirables, les nouveaux serviraient à les activer. Ce changement vers des hypothèses conservatrices résulterait en des oublis nettement moins impactant du point de vue de la sûreté.

Enfin, intégrer des techniques « à la RUSTInA » directement dans les compilateurs permettrait de lever la limitation technique principale : l'ignorance de la sémantique du bloc. Le compilateur serait alors en mesure de vérifier le comportement du bloc et d'avertir l'utilisateur en cas de problème de conformité ou au contraire, d'optimisation manquée. La technique est suffisamment rapide pour ne pas ralentir significativement le temps de compilation. De plus, les connaissances du compilateur au point d'application (propagation de constantes, registres allouables, etc.) pourraient enrichir et spécialiser les analyses du bloc assembleur.

Annexes

Annexe A

Exemple et motivation – codes complets

Nous donnons le code complet de la configuration des différents exemples de la section 2.3. La macro `__FD_ZERO` est redéfinie en C pour le besoin des analyseurs – pour obtenir le comportement bloquant, il suffit de supprimer la ligne qui contient `#undef __FD_ZERO` ainsi que la définition qui suit.

A.1 Main KLEE

```
typedef /* long */ int __fd_mask;
#define __NFDBITS (8 * (int) sizeof (__fd_mask))
#define        __FD_ELT(d)        ((d) / __NFDBITS)
#define __FD_MASK(d) ((__fd_mask) (1UL << ((d) % __NFDBITS)))
#define        __FD_SETSIZE                1024
typedef struct
  {
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->__fds_bits)
  } fd_set;
#define __FD_SET(d, set) \
((void) (__FDS_BITS (set)[__FD_ELT (d)] |= __FD_MASK (d)))
#define __FD_ISSET(d, set) \
((__FDS_BITS (set)[__FD_ELT (d)] & __FD_MASK (d)) != 0)

#  define __FD_ZERO_STOS "stosl"
# define __FD_ZERO(fdsp) \
  do {                                                                  \
    int __d0, __d1;                                                     \
    __asm__ __volatile__ ("cld; rep; " __FD_ZERO_STOS                   \
                          : "=c" (__d0), "=D" (__d1)                    \
                          : "a" (0), "0" (sizeof (fd_set)               \
                                          / sizeof (__fd_mask)),        \
                            "1" (&__FDS_BITS (fdsp)[0])                 \
                          : "memory");                                  \
  } while (0)
```

```
#undef __FD_ZERO
#define __FD_ZERO(fdsp)                                                    \
  do {                                                                     \
    for (int i = 0; i < sizeof (fd_set) / sizeof (__fd_mask); i += 1)      \
      __FDS_BITS(fdsp)[i] = 0;                                             \
  } while (0)

#define FD_SETSIZE __FD_SETSIZE
#define FD_ZERO __FD_ZERO
#define FD_SET __FD_SET
#define FD_ISSET __FD_ISSET

int prepareForSelect(int *socks, int nr, fd_set *read_set) {
    int i;
    int maxFd;
    FD_ZERO(read_set);
    maxFd=-1;
    for(i=0; i<nr; i++) {
        if(socks[i] == -1)
            continue;
        FD_SET(socks[i], read_set);
        if(socks[i] > maxFd)
            maxFd = socks[i];
    }
    return maxFd;
}

#include <klee/klee.h>

#define N 10

int main (int argc, char *argv[])
{
  int socks[N];
  int nr;
  fd_set read_set;

  klee_make_symbolic(&nr, sizeof(nr), "nr");
  klee_assume((0 < nr) & (nr <= N));
  klee_make_symbolic(&socks, sizeof(socks), "socks");
  for (int i = 0; i < N; i += 1)
    klee_assume((-1 <= socks[i]) & (socks[i] < FD_SETSIZE));
  prepareForSelect(socks, nr, &read_set);
  for (int i = 0; i < nr; i += 1)
```

```
    if (socks[i] != -1)
      klee_assert(FD_ISSET(socks[i], &read_set));
  return 0;
}
```

A.2 Main EVA

```
typedef /* long */ int __fd_mask;
#define __NFDBITS (8 * (int) sizeof (__fd_mask))
#define        __FD_ELT(d)        ((d) / __NFDBITS)
#define __FD_MASK(d) ((__fd_mask) (1UL << ((d) % __NFDBITS)))
#define        __FD_SETSIZE                1024
typedef struct
  {
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->__fds_bits)
  } fd_set;
#define __FD_SET(d, set) \
((void) (__FDS_BITS (set)[__FD_ELT (d)] |= __FD_MASK (d)))
#define __FD_ISSET(d, set) \
((__FDS_BITS (set)[__FD_ELT (d)] & __FD_MASK (d)) != 0)

#  define __FD_ZERO_STOS "stosl"
# define __FD_ZERO(fdsp) \
  do {                                                                  \
    int __d0, __d1;                                                     \
    __asm__ __volatile__ ("cld; rep; " __FD_ZERO_STOS                   \
                          : "=c" (__d0), "=D" (__d1)                    \
                          : "a" (0), "0" (sizeof (fd_set)               \
                                          / sizeof (__fd_mask)),        \
                            "1" (&__FDS_BITS (fdsp)[0])                 \
                          : "memory");                                  \
  } while (0)

#undef __FD_ZERO
#define __FD_ZERO(fdsp)                                                 \
  do {                                                                  \
    for (int i = 0; i < sizeof (fd_set) / sizeof (__fd_mask); i += 1)   \
      __FDS_BITS(fdsp)[i] = 0;                                          \
  } while (0)

#define FD_SETSIZE __FD_SETSIZE
#define FD_ZERO __FD_ZERO
```

```
#define FD_SET __FD_SET
#define FD_ISSET __FD_ISSET

int prepareForSelect(int *socks, int nr, fd_set *read_set) {
    int i;
    int maxFd;
    FD_ZERO(read_set);
    maxFd=-1;
    for(i=0; i<nr; i++) {
        if(socks[i] == -1)
            continue;
        FD_SET(socks[i], read_set);
        if(socks[i] > maxFd)
            maxFd = socks[i];
    }
    return maxFd;
}

#include <__fc_builtin.h>

#define N 10

int main (int argc, char *argv[])
{
  int socks[N];
  int nr;
  fd_set read_set;

  nr = Frama_C_interval(1, N);
  for (int i = 0; i < N; i += 1)
    socks[i] = Frama_C_interval(-1, FD_SETSIZE - 1);
  prepareForSelect(socks, nr, &read_set);
  return 0;
}
```

A.3 Annotations de WP

```
typedef /* long */ int __fd_mask;
#define __NFDBITS (8 * (int) sizeof (__fd_mask))
#define        __FD_ELT(d)        ((d) / __NFDBITS)
#define __FD_MASK(d) ((__fd_mask) (1UL << ((d) % __NFDBITS)))
#define        __FD_SETSIZE                1024
typedef struct
```

```
  {
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->__fds_bits)
  } fd_set;
#define __FD_SET(d, set) \
((void) (__FDS_BITS (set)[__FD_ELT (d)] |= __FD_MASK (d)))
#define __FD_ISSET(d, set) \
((__FDS_BITS (set)[__FD_ELT (d)] & __FD_MASK (d)) != 0)

#  define __FD_ZERO_STOS "stosl"
# define __FD_ZERO(fdsp) \
  do {                                                                  \
    int __d0, __d1;                                                     \
    __asm__ __volatile__ ("cld; rep; " __FD_ZERO_STOS                   \
                          : "=c" (__d0), "=D" (__d1)                    \
                          : "a" (0), "0" (sizeof (fd_set)               \
                                          / sizeof (__fd_mask)),        \
                            "1" (&__FDS_BITS (fdsp)[0])                 \
                          : "memory");                                  \
  } while (0)

/* #undef __FD_ZERO */
/* #define __FD_ZERO(fdsp)                                                 \ */
/*     for (int i = 0; i < sizeof (fd_set) / sizeof (__fd_mask); i += 1)   \ */
/*       __FDS_BITS(fdsp)[i] = 0; */

/* #define FD_SETSIZE __FD_SETSIZE */
/* #define FD_ZERO __FD_ZERO */
/* #define FD_SET __FD_SET */
/* #define FD_ISSET __FD_ISSET */

/*@
  requires \valid_read(socks + (0 .. nr - 1));
  requires \valid(read_set);
  requires \forall integer i, j;
    0 <= i < nr && 0 <= j < sizeof (fd_set) / sizeof (__fd_mask)
    ==> socks + i != __FDS_BITS(read_set) + j;
  requires 0 < nr <= FD_SETSIZE ;
  requires \forall integer i;
    0 <= i < nr ==> -1 <= socks[i] < FD_SETSIZE;
  assigns *(__FDS_BITS(read_set)
            + (0 .. sizeof (fd_set) / sizeof (__fd_mask) - 1));
 */
int prepareForSelect(int *socks, int nr, fd_set *read_set) {
```

```
    int i;
    int maxFd;
    /*@
      loop invariant \forall integer k;
        0 <= k < i ==> __FDS_BITS(read_set)[k] == 0;
      loop invariant 0 <= i <= sizeof (fd_set) / sizeof (__fd_mask);
      loop invariant \forall integer k;
        0 <= k < nr ==> socks[k] == \at(socks[k], Pre);
      loop assigns i,
        *(__FDS_BITS(read_set)
           + (0 .. sizeof (fd_set) / sizeof (__fd_mask) - 1));
      loop variant sizeof (fd_set) / sizeof (__fd_mask) - i;
    */
    FD_ZERO(read_set);
    maxFd=-1;
    /*@
      loop invariant
        \forall integer k; 0 <= k < nr ==> -1 <= socks[k] < FD_SETSIZE;
      loop invariant 0 <= i <= nr;
      loop assigns maxFd, i,
        *(__FDS_BITS(read_set) +
          (0 .. sizeof (fd_set) / sizeof (__fd_mask) - 1));
      loop variant nr - i;
    */
    for(i=0; i<nr; i++) {
        if(socks[i] == -1)
            continue;
        FD_SET(socks[i], read_set);
        if(socks[i] > maxFd)
            maxFd = socks[i];
    }
    return maxFd;
}
```

Annexe B

Règles de réécriture syntaxiques

Nous listons dans cette annexe l'ensemble des règles de réécriture présentes dans l'outil BINSEC :

— propagation de constantes (arithmétique modulaire)

— éléments neutres :

$$x + 0 \hookrightarrow x \qquad x \wedge \vec{1}_{\texttt{sizeof}(x)} \hookrightarrow x$$
$$x - 0 \hookrightarrow x \qquad x \vee 0 \hookrightarrow x$$
$$x \times 1 \hookrightarrow x \qquad x \oplus 0 \hookrightarrow x$$
$$x \operatorname{udiv} 1 \hookrightarrow x \qquad x \operatorname{shl} 0 \hookrightarrow x$$
$$x \operatorname{sdiv} 1 \hookrightarrow x \qquad x \operatorname{shr} 0 \hookrightarrow x$$
$$x \operatorname{urem} 2^{\texttt{sizeof}(x)} \hookrightarrow x \qquad x \operatorname{sar} 0 \hookrightarrow x$$

— idempotence :

$$x \wedge x \hookrightarrow x \qquad \operatorname{sext}_{\texttt{sizeof}(x)}(x) \hookrightarrow x$$
$$x \vee x \hookrightarrow x \qquad \operatorname{extract}_{0..\texttt{sizeof}(x)-1}(x) \hookrightarrow x$$
$$\operatorname{uext}_{\texttt{sizeof}(x)}(x) \hookrightarrow x$$

— éléments absorbants :

$$x \times 0 \hookrightarrow 0 \qquad x \operatorname{urem} 1 \hookrightarrow 0$$
$$x \wedge 0 \hookrightarrow 0 \qquad x \operatorname{srem} 1 \hookrightarrow 0$$
$$x \vee 1 \hookrightarrow 1$$

— éliminations :

$$x - x \hookrightarrow 0 \qquad x \oplus x \hookrightarrow 0$$
$$x \operatorname{udiv} x \hookrightarrow 1 \qquad x \operatorname{sdiv} x \hookrightarrow 1$$
$$x \operatorname{shl} k \overset{\texttt{sizeof}(x) \leq k}{\hookrightarrow} 0 \qquad x \operatorname{shr} k \overset{\texttt{sizeof}(x) \leq k}{\hookrightarrow} 0$$

— involutions :

$$\neg(\neg x) \hookrightarrow x \qquad -(-x) \hookrightarrow x$$

— factorisation d'opérandes :

$$(x \operatorname{shl} y) \operatorname{shl} z \hookrightarrow x \operatorname{shl} (y + z)$$
$$(x \operatorname{shr} y) \operatorname{shr} z \hookrightarrow x \operatorname{shr} (y + z)$$
$$(x \operatorname{sar} y) \operatorname{sar} z \hookrightarrow x \operatorname{sar} (y + z)$$

— suppression d'opérations redondantes :

$$(x \operatorname{urem} k) \operatorname{urem} k' \overset{k \leq k'}{\hookrightarrow} x \operatorname{urem} k$$
$$(x \operatorname{urem} k) \operatorname{urem} k' \overset{k > k'}{\hookrightarrow} x \operatorname{urem} k'$$
$$\operatorname{sext}_k(\operatorname{uext}_{k'}(x)) \overset{k' > \texttt{sizeof}(x)}{\hookrightarrow} \operatorname{uext}_k(x)$$
$$\operatorname{uext}_k(\operatorname{uext}_{k'}(x)) \hookrightarrow \operatorname{uext}_k(x)$$

— simplifications de conditions

$$C = 1 \hookrightarrow C \qquad x >_u x \hookrightarrow 0$$
$$C \neq 0 \hookrightarrow C \qquad x <_u x \hookrightarrow 0$$
$$C > 0 \hookrightarrow C \qquad x >_s x \hookrightarrow 0$$
$$x = x \hookrightarrow 1 \qquad x <_s x \hookrightarrow 0$$
$$x \neq x \hookrightarrow 0$$

— loi de De Morgan (étendue)

$$\neg(C \wedge C') \hookrightarrow \neg C \vee \neg C' \qquad \neg(x >_u y) \hookrightarrow y \geq_u x$$
$$\neg(C \vee C') \hookrightarrow \neg C \wedge \neg C' \qquad \neg(x \geq_u y) \hookrightarrow y >_u x$$
$$x \oplus \vec{1}_{\texttt{sizeof}(x)} \hookrightarrow \neg x \qquad \neg(x <_s y) \hookrightarrow y \leq_s x$$
$$\neg(x = y) \hookrightarrow x \neq y \qquad \neg(x \leq_s y) \hookrightarrow y <_s x$$
$$\neg(x \neq y) \hookrightarrow x = y \qquad \neg(x >_s y) \hookrightarrow y \geq_s x$$
$$\neg(x <_u y) \hookrightarrow y \leq_u x \qquad \neg(x \geq_s y) \hookrightarrow y >_s x$$
$$\neg(x \leq_u y) \hookrightarrow y <_u x$$

— simplification d'expressions ternaires :

$$C \ ?\ false : true \hookrightarrow \neg C$$
$$\neg C \ ?\ x : y \hookrightarrow C \ ?\ y : x$$
$$C \ ?\ true : false \hookrightarrow C$$
$$C \ ?\ x : x \hookrightarrow x$$
$$x \diamond (C \ ?\ y : z) \hookrightarrow C \ ?\ x \diamond y : x \diamond z$$
$$(C \ ?\ w : x) \diamond (C \ ?\ y : z) \hookrightarrow C \ ?\ w \diamond y : x \diamond z$$

— séparations d'(in-)égalités :

$$k = \text{concat}(x, y) \hookrightarrow (\text{extract}_{\texttt{sizeof}(y)..\texttt{sizeof}(x)+\texttt{sizeof}(y)-1}(k) = x) \wedge (\text{extract}_{0..\texttt{sizeof}(y)-1}(k) = y)$$
$$k \neq \text{concat}(x, y) \hookrightarrow (\text{extract}_{\texttt{sizeof}(y)..\texttt{sizeof}(x)+\texttt{sizeof}(y)-1}(k) \neq x) \vee (\text{extract}_{0..\texttt{sizeof}(y)-1}(k) \neq y)$$

— reconstruction de concaténations :

$$\text{uext}_{\texttt{sizeof}(x)+\texttt{sizeof}(y)}(x) \vee \text{concat}(y, 0_{\texttt{sizeof}(x)}) \hookrightarrow \text{concat}(y, x)$$
$$\text{uext}_{\texttt{sizeof}(x)+k}(x)\ \text{shl}\ k \hookrightarrow \text{concat}(x, 0_k)$$
$$\text{concat}(0_k, x) \hookrightarrow \text{uext}_{k+\texttt{sizeof}(x)}(x)$$

— simplifications d'extractions :

$$\begin{aligned}
\mathrm{extract}_{i..j}(\mathrm{extract}_{k..l}(x)) &\hookrightarrow \mathrm{extract}_{i+k..j+k}(x)\\
\mathrm{concat}(\mathrm{extract}_{i..j}(x), \mathrm{extract}_{j+1..k}(x)) &\hookrightarrow \mathrm{extract}_{i..k}(x)\\
\mathrm{extract}_{i..j}(\mathrm{uext}_k(x)) &\overset{\mathtt{sizeof}(x)\leq i}{\hookrightarrow} 0\\
\mathrm{extract}_{0..\mathtt{sizeof}(x)-1}(x) &\hookrightarrow x\\
\mathrm{extract}_{i..j}(\mathrm{concat}(x,y)) &\overset{j<\mathtt{sizeof}(y)}{\hookrightarrow} \mathrm{extract}_{i..j}(y)\\
\mathrm{extract}_{i..j}(\mathrm{concat}(x,y)) &\overset{\mathtt{sizeof}(y)\leq i}{\hookrightarrow} \mathrm{extract}_{i-\mathtt{sizeof}(y)..j-\mathtt{sizeof}(y)}(x)\\
\mathrm{extract}_{0..j}(\mathrm{uext}_k(x)) &\overset{\mathtt{sizeof}(x)\leq j}{\hookrightarrow} \mathrm{uext}_j(x)\\
\mathrm{extract}_{i..j}(\mathrm{uext}_k(x)) &\overset{j<\mathtt{sizeof}(x)}{\hookrightarrow} \mathrm{extract}_{i..j}(x)\\
\mathrm{extract}_{0..j}(\mathrm{sext}_k(x)) &\overset{\mathtt{sizeof}(x)\leq j}{\hookrightarrow} \mathrm{sext}_j(x)\\
\mathrm{extract}_{i..j}(\mathrm{sext}_k(x)) &\overset{j<\mathtt{sizeof}(x)}{\hookrightarrow} \mathrm{extract}_{i..j}(x)
\end{aligned}$$

— reconstruction d'opérations arithmétique en complément à 2 :

$$\begin{aligned}
\neg x + 1 &\hookrightarrow -x\\
(\mathrm{uext}_k(x) \oplus 2^{\mathtt{sizeof}(x)-1}) - 2^{\mathtt{sizeof}(x)-1} &\hookrightarrow \mathrm{sext}_k(x)\\
\mathrm{extract}_{\mathtt{sizeof}(x)-1}(x) &\hookrightarrow x <_s 0\\
\mathrm{uext}_n(C) - 1 &\hookrightarrow C \ ?\ 0 : \vec{1}_n\\
\mathrm{sext}_n(C) &\hookrightarrow C \ ?\ \vec{1}_n : 0
\end{aligned}$$

Bibliographie

[Att07] Christian Attiogbé. *Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette*. Habilitation à diriger des recherches, Université de Nantes, September 2007. (Cité en page 2.)

[BBCD18] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *WP Manual*, Frama-C Argon-20181129 edition, 2018. (Cité en page 111.)

[BBW14] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A precise and abstract memory model for c using symbolic values. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 449–468, Cham, 2014. Springer International Publishing. (Cité en pages 43, 47 et 61.)

[BBW17] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS : A Memory-Aware Verified C Compiler using Pointer as Integer Semantics. In *ITP 2017 - 8th International Conference on Interactive Theorem Proving*, volume 10499 of *ITP 2017 : Interactive Theorem Proving*, pages 81–97, Brasilia, Brazil, September 2017. Springer. (Cité en page 61.)

[BBY17] Sandrine Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *18th International Conference on Verification Model Checking and Abstract Interpretation (VMCAI 2017)*, volume 10145 LNCS of *Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation*, pages 112–130, Paris, France, January 2017. (Cité en page 19.)

[BCF+20] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2020. (Cité en page 21.)

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier : Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004. (Cité en page 28.)

[BDM17] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE : targeting infeasibility questions on obfuscated codes. In *International Symposium on Security & Privacy (S&P'17)*. IEEE, 2017. (Cité en page 90.)

[BGM13] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints : Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131, 2013. (Cité en page 30.)

[BH08] S. Bardin and P. Herrmann. Structural testing of executables. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 22–31, 2008. (Cité en page 61.)

[BHL+11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The BINCOA framework for binary code analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 165–170. Springer, 2011. (Cité en pages 41, 43 et 45.)

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011. (Cité en page 32.)

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap : A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cité en pages 5, 32, 41, 43 et 120.)

[BLSW13] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 353–368. USENIX Association, 2013. (Cité en page 120.)

[Boe15] S. Boessenkool. Bug 68095 – comment 4, 2015. (Cité en pages 88 et 89.)

[BR10] Gogul Balakrishnan and Thomas W. Reps. WYSINWYX : what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6) :23 :1–23 :84, 2010. (Cité en pages 32 et 61.)

[BRA10] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010. (Cité en page 121.)

[Büh17] David Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions :EVA, an Evolved Value Analysis for Frama-C*. PhD thesis, University of Rennes 1, France, 2017. (Cité en page 111.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. (Cité en pages 28 et 30.)

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, LNCS #3444, page 21. Springer, 2005. (Cité en pages 2 et 28.)

[CCF18] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception : System-wide security testing of real-world embedded systems software. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 309–326. USENIX Association, 2018. (Cité en pages 5, 62, 92, 96 et 120.)

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE : unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. (Cité en pages 2, 19, 20, 30 et 111.)

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc : A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics : 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cité en page 119.)

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *IN POPL*, pages 289–300, 2009. (Cité en page 2.)

[CdSSPT14] Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, and Aaron Tomb. Formal verification of klibc with the WP frama-c plug-in. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2014. (Cité en page 28.)

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. (Cité en page 28.)

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, October 1991. (Cité en page 100.)

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018. (Cité en page 2.)

[CKOS04] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 85–96, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (Cité en page 30.)

[CMPP11] Pascal Cuoq, B. Monate, Anne Pacalet, and V. Prevosto. Functional dependencies of c functions via weakest pre-conditions. *International Journal on Software Tools for Technology Transfer*, 13 :405–417, 2011. (Cité en page 19.)

[Cou96] Patrick Cousot. Program analysis : The abstract interpretation perspective. *ACM Comput. Surv.*, 28(4es) :165–es, December 1996. (Cité en pages 2 et 21.)

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013. (Cité en page 31.)

[d'A20] Amanieu d'Antras. Inline assembly #2873, 2020. (Cité en page 126.)

[DB15] Adel Djoudi and Sébastien Bardin. Binsec : Binary code analysis with low-level regions. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–217, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. (Cité en pages 5, 32, 43, 47, 59 et 61.)

[DBF+16] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specifi-

cation of concretization and symbolization policies in symbolic execution. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 2016. (Cité en page 90.)

[DBG16] Adel Djoudi, Sébastien Bardin, and Éric Goubault. Recovering high-level conditions from binary programs. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016 : Formal Methods*, pages 235–253, Cham, 2016. Springer International Publishing. (Cité en page 100.)

[DBR20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel : Efficient relational symbolic execution for constant-time at binary-level. In *International Symposium on Security and Privacy (SP'20)*. IEEE, 2020. (Cité en page 90.)

[DBR21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter — efficient relational symbolic execution for spectre with haunted relse. In *The Network and Distributed System Security Symposium (NDSS)*, 2021. (Cité en page 90.)

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, August 1975. (Cité en pages 2 et 21.)

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. (Cité en page 28.)

[FHRS08] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried. Some assembly required - program analysis of embedded system code. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 15–24, Sept 2008. (Cité en pages 4, 62, 91 et 124.)

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society. (Cité en page 28.)

[FMB+16] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap : combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016*. ACM, 2016. (Cité en page 90.)

[GCC20] GCC. Extended asm - assembler instructions with c expression operands, 2020. (Cité en pages 36 et 58.)

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : Directed automated random testing. *SIGPLAN Not.*, 40(6) :213–223, June 2005. (Cité en page 28.)

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th*

International Conference on Model Checking Software, SPIN'03, page 235–239, Berlin, Heidelberg, 2003. Springer-Verlag. (Cité en page 28.)

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969. (Cité en page 28.)

[KFJ+17] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 353–364. IEEE Computer Society, 2017. (Cité en page 90.)

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery. (Cité en pages 32 et 66.)

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c : A software analysis perspective. *Formal Asp. Comput.*, 27(3) :573–609, 2015. (Cité en pages 2, 28 et 58.)

[KT14] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014. (Cité en page 28.)

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2009. (Cité en page 61.)

[LAB11] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie : Principled reverse engineering of types in binary programs. In *NDSS*, 2011. (Cité en page 31.)

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1) :1–31, 2008. (Cité en pages 41, 43 et 61.)

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, 2009. (Cité en page 2.)

[LQ08] Shuvendu Lahiri and Shaz Qadeer. Back to the future : Revisiting precise program verification using smt solvers. *SIGPLAN Not.*, 43(1) :171–182, January 2008. (Cité en page 30.)

[Mau11] Stefan Maus. *Verification of hypervisor subroutines written in Assembler*. PhD thesis, University of Freiburg, Germany, 2011. (Cité en pages 3, 5 et 119.)

[McC07] D. McCall. Use of input/output operands in __asm__ templates not fully documented, 2007. (Cité en page 37.)

[MGS08] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *2008 Formal Methods in Computer-Aided Design*, pages 1–8, Nov 2008. (Cité en page 120.)

[MHH+19] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing : A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. (Cité en page 107.)

[MM16] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 24–35. ACM, 2016. (Cité en page 32.)

[MMS08] Stefan Maus, Michał Moskal, and Wolfram Schulte. Vx86 : x86 assembler simulated in c powered by automated theorem proving. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, pages 284–298, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. (Cité en pages 55, 61, 96 et 119.)

[Mun12] Patrick Munier. *Polyspace®*, pages 123–153. ISTE Ltd, 2012. (Cité en page 2.)

[MWCG98] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97. ACM, 1998. (Cité en page 120.)

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 83–94. ACM, 2000. (Cité en page 121.)

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. (Cité en page 30.)

[PCR+17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages : How do energy, time, and memory relate ? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery. (Cité en pages 1 et 10.)

[PM12] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification : LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 45–95, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cité en page 30.)

[PPDA17] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. Funtal : Reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 495–509, New York, NY, USA, 2017. ACM. (Cité en page 120.)

[RBB+19] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M. Potet. Get rid of inline assembly through verification-oriented lifting. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 577–589, Nov 2019. (Cité en pages 7 et 59.)

[RBB+21] F. Recoules, S. Bardin, R. Bonichon, M. Lemerre, L. Mounier, and M. Potet. Interface compliance of inline assembly : Automatically check, patch and refine. In *43th International Conference on Software Engineering (ICSE)*, May 2021. (Cité en page 7.)

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953. (Cité en page 2.)

[Riv04] Xavier Rival. Certification of compiled assembly code by invariant translation. *STTT*, 6(1) :15–37, 2004. (Cité en page 121.)

[RKS16] Ed Robbins, Andy King, and Tom Schrijvers. From minx to minc : Semantics-driven decompilation of recursive datatypes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 191–203, New York, NY, USA, 2016. ACM. (Cité en page 31.)

[RMK+18] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An analysis of x86-64 inline assembly in c

programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pages 84–99, New York, NY, USA, 2018. ACM. (Cité en pages 3 et 39.)

[Sch08] Wolfram Schulte. Vcc : Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, January 2008. (Cité en page 5.)

[SMK13] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013. (Cité en page 121.)

[SRN+18] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Logino. Evolving exact decompilation. In *BAR 2018, Workshop on Binary Analysis Research, San Diego, California, USA, February 18, 2018*, 2018. (Cité en page 120.)

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software : Theories, Tools, Experiments : 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, pages 18–33, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cité en page 119.)

[Sta18] O. Stannard. [llvm-dev] [rfc] checking inline assembly for validity, November 2018. (Cité en page 13.)

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016. (Cité en pages 5, 32, 41 et 43.)

[Ven12] Arnaud J. Venet. The gauge domain : Scalable analysis of linear inequality invariants. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 139–154, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cité en page 105.)

[VPK04] Willem Visser, Corina S. Pundefinedsundefinedreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, page 97–107, New York, NY, USA, 2004. Association for Computing Machinery. (Cité en page 28.)