De l'assembleur sur la ligne ? Appelez TInA !

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

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

Résumé Les méthodes formelles appliquées au logiciel ont fait des progrès importants lors des deux dernières décennies. Leur application à l'embarqué est ainsi un succès indéniable. Parmi les prochains défis réside la question de leur application à des domaines moins contraints. Par exemple, le développement en C de logiciels non-critiques utilise régulièrement l'insertion d'assembleur « en ligne », que ce soit pour optimiser certaines opérations ou pour accéder à des primitives systèmes autrement inaccessibles. Ceci empêche complètement l'utilisation des méthodes développés pour le C pur. Nous proposons ici TInA, une méthode générale, automatique, sûre et orientée vérification pour pouvoir porter le code assembleur en ligne vers un code C sémantiquement équivalent, et profiter en retour des analyses pré-existantes pour ce langage. Nos expérimentations sur du code C d'envergure montre la faisabilité et l'intérêt de l'approche.

1 Introduction

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

Problème. Nous considérons ici le problème de l'analyse de code « mixte », combinant assembleur en ligne et code C/C++. Cette fonctionnalité, présente dans les compilateurs GCC, clang et Visual Studio, permet d'intégrer des instructions assembleur au sein d'un programme C/C++. Elle est utilisée en général pour des raisons d'efficacité ou d'accès à des fonctionnalités bas niveau qui ne peuvent être déclenchées depuis le langage hôte. L'usage d'assembleur en ligne est plus fréquent qu'on n'imagine : ainsi **11%** des paquets Debian 8.11 écrits en C/C++ utilisent de l'assembleur en ligne, directement ou à travers des bibliothèques, avec

des blocs allant jusqu'à 500 instructions, et **28%** des projets C les plus populaires sur Github en contiennent, d'après Rigger et coll. [35]. En vérité, l'assembleur en ligne est utilisé fréquemment dans des domaines comme la cryptographie, le multimédia ou les pilotes matériel.

Cependant, les analyseurs de programme C/C++ gèrent mal cette fonctionnalité. Certains comme Frama-C [28] la gèrent peu, d'autres comme KLEE [11] ne le gèrent pas du tout. Dans ce cas, l'analyse produira des résultats incorrects ou incomplets. Ceci correspond à un réel problème d'applicabilité des techniques avancées d'analyse de code.

Étant donné qu'il est très coûteux de redévelopper des analyseurs dédiés, la façon usuelle de traiter ces morceaux de code assembleur est de proposer un code équivalent, dans le langage hôte (C/C++) ou bien sous forme de spécification logique — deux méthodes que Frama-C [28] permet. Dans les deux cas, cette tâche est effectuée manuellement : cela met d'emblée hors de portée l'analyse de larges bases de code. Il est en effet chronophage d'annoter ou de traduire manuellement de gros bouts d'assembleur, et cela est aussi une source d'erreur non négligeable. Plus le morceau d'assembleur est gros, plus ces problèmes peuvent devenir importants.

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

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

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

Compatible avec la vérification formelle La traduction se doit d'être agnostique vis-à-vis des techniques de vérifications classiques (e.g., exécution symbolique [27], vérification déductive [23,24] ou interprétation abstraite [17]), tout en permettant une qualité d'analyse post-traduction suffisamment bonne en pratique (nous utiliserons pour décrire cette propriété le terme de *vérifiabilité*).

Les travaux menés jusqu'à présent en vérification de code « mixte » ne remplissent pas tous ces objectifs. Ainsi, Vx86 [30] est ciblé `x86` (architecture Intel 32-bits) et vérification déductive mais ne fournit aucun moyen de montrer la correction de la traduction. Au premier abord, les techniques de décompilation [14,**?**,**?**] peuvent sembler bien adaptées. Mais leur but premier est l'aide à la rétro-ingénierie. De fait, cette famille de techniques peut remettre en cause la correction du processus, à tel point que « les décompilateurs existants produisent fréquemment un code décompilé qui n'est pas complètement fonctionnellement équivalent au programme d'origine » [37]. Certains travaux récents [9,37] ciblent

ce critère de correction mais Schwartz et. al [9] n'étudient pas la question de savoir si le code produit peut être vérifié par les outils actuels et ne démontrent pas la correction de leur approche — bien qu'à leur crédit, ils montrent un certain degré de correction via du test intensif. Schulte et coll. [37] démontre sans équivoque la correction de leur méthode mais l'approche *search-based* qu'ils proposent ne termine pas toujours et ne parle pas de vérifiabilité.

Proposition. Nous proposons TInA (*Taming Inline Assembly*), une technique automatique, générique, sûre, orientée vérification pour porter du code assembleur en ligne vers un C équivalent.

Un élément clé de TInA est qu'en nous concentrant sur l'assembleur en ligne, plutôt que sur le problème général de la décompilation, nous attaquons un problème plus restreint (taille de code, portée bien définie, type d'instructions et de constructions), mieux défini (grâce à l'interface avec le code C), ouvrant ainsi la porte à une technique ciblée plus puissante.

En particulier, TInA est fondé sur les principes suivants :

— La réutilisation de plateformes d'analyse de code exécutable [8,19,26,**?**] qui permettent de porter du code binaire vers un langage intermédiaire, indépendant de l'architecture matérielle, éprouvé et concis.

— La validation de la traduction, pour la sûreté de notre méthode, plutôt que la validation du traducteur. Ce problème est en général plus aisé et réduit la base de confiance à un vérificateur testé intensivement, et qui pourrait, lui, être validé.

— Un algorithme de vérification d'équivalence de programmes dédié [3], ramené ici à la résolution d'un problème plus réduit, ciblé pour notre processus.

— Des passes de transformations dédiées, pour améliorer la vérifiabilité de notre résultat, par raffinements successifs du langage intermédiaire brut vers des abstractions le rapprochant de C (flot de contrôle haut niveau, types de données, opérations).

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

— Une chaîne outillée permettant la vérification de programme mélangeant de l'assembleur en ligne et du C (Sec. 3.1) ;

— Une méthode (Sec. 3.2) pour ramener l'assembleur en ligne à du code C, améliorée par l'usage du contexte dans lequel l'assembleur se trouve inclus, et validée automatiquement afin de pouvoir faire confiance au processus de traduction mais également de ne pas mettre en péril les critères de correction des analyses formelles considérées ;

— Le prototypage de notre méthode atteste de son intérêt pratique (Sec. 4) : nous portons et validons 100% des blocs assembleur de 3 projets libres importants, et nous confirmons aussi la vérifiabilité du code produit.

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

2 Contexte et motivation

Principe. Concentrons-nous sur un extrait de code assembleur en ligne (x86) décrit en Fig. 1a, provenant des sources du logiciel UDPCast. Le compilateur va traiter ce morceau quasiment en aveugle. En réalité, il n'est concerné que par la spécification des entrées, sorties et éléments modifiés (en anglais, *clobbers*), donnée par le programmeur et contenant des contraintes qui pourront par exemple être utilisées lors de l'allocation de registres. Ce mode d'utilisation avec spécification, représentatif de l'assembleur en ligne dit « étendu », est celui que le manuel de GCC conseille. Mais le code assembleur en lui-même, notamment toutes ses mnémoniques, est opaque et est propagé *tel quel* — en tant que chaîne de caractères — jusqu'à l'émission de code.

```
# 54 "/usr/include/i386-linux-gnu/sys/select.h"
typedef long int __fd_mask;

# 64 "/usr/include/i386-linux-gnu/sys/select.h"
typedef struct {
  __fd_mask __fds_bits[1024 / (8 * sizeof(__fd_mask))];
} fd_set;

# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  do {
    int __d0, __d1;
    __asm__ __volatile__
      ("cld; rep; " "stosl"
       : "=c" (__d0), "=D" (__d1)
       : "a" (0),
         "0" (sizeof(fd_set) / sizeof(__fd_mask)),
         "1" (&((read_set)->__fds_bits)[0])
       : "memory");
  } while (0);
  /* [...] */
  return maxFd;
}
```

(a) Version originale (socklib.i)

```
# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  {
    int __d0;
    int __d1;
    __fd_mask *__tina_4;
    unsigned int __tina_3;
    __tina_3 = sizeof(fd_set) / sizeof(__fd_mask);
    __tina_4 = & read_set->__fds_bits[0];
    {
      long *__tina_edi;
      unsigned int __tina_ecx;
      __TINA_BEGIN_1__: ;
      __tina_ecx = __tina_3;
      __tina_edi = __tina_4;
      while (0U != __tina_ecx) { // rep
         *__tina_edi = 0;        // stosl
         __tina_edi ++;          // rep
         __tina_ecx --;          // rep
      }
      __TINA_END_1__: ;
    }
  }
  /* [...] */
  return maxFd;
}
```

(b) Version C générée par TInA

FIGURE 1: Exemple initial

Exemple. Le code en Fig. 1a est entouré d'une spécification, à partir d'un langage concis de description de contraintes, dans des zones séparées par ' :'.

— Tout d'abord des contraintes d'allocations de variables pour les sorties :

0. `"=c" (__d0)` indique que la variable `___d0` doit être affectée au registre ecx ;
1. `"=D" (__d1)` indique que la variable `___d1` doit être affectée au registre edi ;

En fait ces contraintes sont présentes ici pour pouvoir y référer dans la partie suivante tout en explicitant au compilateur que la valeur contenue par le

registre (par exemple ecx) à la fin du bloc peut être différente de celle passée en entrée.

— Nous avons ensuite la description des entrées : "a" (eax) contient la valeur 0, le registre décrit en 0 (i.e. ecx) contient `sizeof(fd_set) / sizeof(__fd_mask)` et celui en 1 (edi) contient `(&((read_set)->__fds_bits)[0])`.

— Enfin, il est spécifié que toute la mémoire peut être changée ("memory"). Cela indique au compilateur de sauver ce qu'il juge précieux de sa mémoire avant d'entrer dans ce bloc.

Cette spécification s'applique au code "cld ; rep ;" "stosl" dont la sémantique est qu'il met à zéro l'indicateur (*flag*) de direction df, puis remplit ecx double-mots à partir du pointeur edi avec la valeur contenue dans eax. Comme précisé dans le manuel de l'architecture [25], df dirige le signe de l'incrément : quand df est à zéro, l'incrément est positif. Le code produit par TInA est donné en Fig. 1b : il comporte bien une boucle, que la sémantique informelle annonçait, mais des éléments (cld, "a" (0)) ont été nettoyés du résultat.

Comportement des analyses. Si nous essayons d'analyser du code comportant ce genre de morceau d'assembleur en ligne, les outils peuvent avoir des comportements erratiques. Certains s'arrêtent tout bonnement avec une erreur — c'est après tout un comportement correct — d'autres, comme Frama-C, émettent un messages d'alarme. Ici Frama-C émet "Clobber list contain "memory" argument. Assuming no side-effect beyond those mentioned in output operands", message clair mais dont la correction est discutable puisque le mot-clé "memory" explicite justement que nous écrivons potentiellement dans toute la mémoire et Frama-C l'ignore en ne prenant en compte que les sorties — le seul choix correct, sans analyser l'assembleur, est de s'arrêter (ou de mettre toute la mémoire à $\top$ si nous étions en inteprétation abstraite). Une seule ligne d'assembleur a ainsi le pouvoir de mettre à mal les outils d'analyse. Bien entendu, on peut contourner le problème, par exemple en réécrivant manuellement le morceau en code C sémantiquement équivalent. Cependant, il est clair que la méthode manuelle aura du mal à passer à l'échelle et sera aussi source d'erreurs.

Propriétés de l'assembleur en ligne. Nous pouvons améliorer cet état de fait en exploitant les propriétés suivantes, spécifiques à l'assembleur en ligne :

P1. La taille des morceaux d'assembleur en ligne est généralement réduite : un tel morceau comporte en moyenne moins de 10 instructions, atteignant rarement plusieurs centaines d'instructions ;

P2. La structure du flot de contrôle est relativement simple, avec seulement quelques conditionnelles et boucles, et aucun saut calculé dynamiquement ;

P3. L'interface des morceaux d'assembleur avec le code C est généralement spécifiée ;

P4. Enfin, le morceau d'assembleur est intégré à un contexte (ici en C), qui contient en particulier des informations de typage : c'est une information particulièrement recherchée en décompilation, souvent de manière heuristique, que nous n'avons dans notre cas plus qu'à propager.

En tirant parti de ces éléments, nous allons définir en Sec. 3 une méthode automatique (pour passer à l'échelle), sûre (par validation de la correction de la technique) et orientée vérification (pour pouvoir continuer à utiliser les outils avancés de vérification C déjà produits) pour obtenir le code C depuis l'assembleur en ligne. De surcroît, cette méthode est générique : elle fonctionne de la même façon quelle que soit l'architecture cible du code assembleur.

3 TInA : porter l'assembleur en ligne vers C

Nous allons décrire TInA tout d'abord dans ses grandes lignes (Sec. 3.1) avant de détailler les phases techniques internes du portage (Sec. 3.2).

3.1 Survol de la méthode

La méthode que nous proposons porte l'assembleur en ligne vers un code C sémantiquement équivalent, en utilisant les propriétés P1–P4 décrites en Sec. 2. Nous procédons en deux temps : tout d'abord, le code est *porté* de l'assembleur vers le C puis cette traduction est *validée*. Nous détaillerons quelques aspects importants de ces phases en Sec. 3.2. Mais concentrons-nous tout d'abord sur l'approche générale, schématisée en Fig. 2.

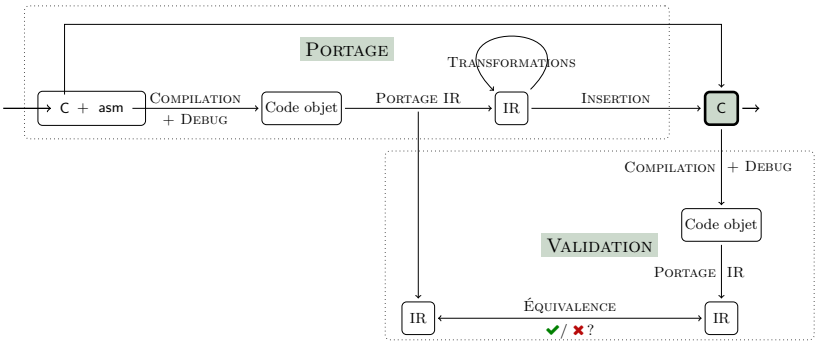

Figure 2: TInA : une vue haut niveau

Notre méthode exploite les fonctionnalités d'outils existants comme Frama-C [28], pour son *front-end* C, et Binsec [19] pour traduire le code exécutable en représentation intermédiaire DBA [3], faire nos analyses et utiliser les solveurs SMT [5] pour la validation.

Localisation. Le processus commence par l'identification d'un code source C qui contient un ou plusieurs morceaux d'assembleur. Ce code initial nous permet de localiser les morceaux d'assembleur via Frama-C — c'est ce qui nous permettra finalement de substituer un code C équivalent au bon endroit.

Nous extrayons également la spécification décrite par le programmeur du morceau d'assembleur : entrées, sorties, zones modifiées. Nous l'utiliserons pour vérifier que le code écrit la respecte. Cette vérification est une passe que les compilateurs ne font pas : en effet, l'assembleur en ligne est propagé comme chaîne de caractères, dans laquelle les motifs identifiés seront remplacés lors de la phase d'allocation de registres (par des registres, des variables de piles ou de tas). Ainsi, *les mnémoniques assembleurs sont gardées intactes par le compilateur.* Par ailleurs, la spécification contient l'information nécessaire pour lier les noms de variable C aux noms de registres ou de zones mémoire dans notre représentation intermédiaire étendue.

Compilation. À la fin de cette phase d'extraction, le code source est compilé pour l'architecture cible *avec les informations de débogage.* Une fois encore, nous allons exploiter la présence du code source pour nous faciliter la tâche. En effet, nous avons la possibilité de contrôler la compilation du code. Par conséquent, nous choisirons bien entendu d'inclure toutes les données qui peuvent nous faciliter la tâche de reconstruction de code C. Par exemple, nous souhaitons avoir accès aux noms et types de variables du contexte. Ici , nous exploitons le format de débogage DWARF [21] pour faire passer cette information jusqu'au binaire duquel nous porterons le code vers du C.

$$
\begin{array}{rl}
\mathsf{inst} & := \mathsf{lv} \leftarrow \mathsf{e} \mid \mathsf{goto\ e} \mid \mathsf{if\ e\ then\ goto\ e} \\
\mathsf{lv} & := \mathsf{var} \mid [\mathsf{e}]_{\mathsf{n}} \\
\mathsf{e} & := \mathsf{cst} \mid \mathsf{var} \mid [\mathsf{e}]_{\mathsf{n}} \mid \mathsf{unop\ e} \mid \mathsf{binop\ e} \\
\mathsf{unop} & := \neg \mid - \mid \mathsf{uext_n} \mid \mathsf{sext_n} \mid \mathsf{restrict_{i\ldots j}} \\
\mathsf{binop} & := + \mid - \mid \times \mid /_{\mathsf{u,s}} \mid \%_{\mathsf{u,s}} \mid \wedge \mid \vee \mid \oplus \\
& \quad \mid \ << \mid >>_{\mathsf{u,s}} \mid >_{\mathsf{u,s}} \mid <_{\mathsf{u,s}} \mid = \mid \mathsf{concat}
\end{array}
$$

FIGURE 3: DBA : un langage intermédiaire bas niveau à base de bit-vecteurs

Portage vers IR. À ce stade, nous commençons le processus de traduction lui-même. D'abord, nous utilisons BINSEC pour revenir du code binaire vers une représentation intermédiaire DBA [3]. L'IR DBA (Fig. 3) est un langage minimaliste bas niveau à la sémantique est bien définie, utilisé dans BINSEC pour modéliser la sémantique des mnémoniques. Il est indépendant de l'architecture (BINSEC supporte x86 et ARM-v7) et ses opérations sont toutes réalisées sur des bit-vecteurs. L'IR DBA, étendu par l'information de débogage de la compilation, servira de support aux différentes passes d'analyses et de raffinements. La mémoire est gérée comme un tableau accédé par l'opérateur [ ].

L'utilisation directe du code binaire peut sembler une complication arbitraire à première vue. C'est en réalité l'unique endroit où le morceau d'assembleur se trouve totalement instancié et plongé dans son contexte — les noms de registre sont explicités, les positions dans la mémoire ont été résolues par le compilateur. Si nous nous étions positionnés dans une phase amont, nous aurions eu à résoudre au

moins un problème similaire à l'allocation de registre — une fois par architecture supportée.

Portage vers C. À cet instant, nous sommes prêts à entamer le portage proprement dit à travers une succession de transformations détaillées en Sec. 3.2, dont l'architecture interne ressemble à celle d'un compilateur. Le résultat est un code C pur débarrassé de l'assembleur en ligne remplacé par un code C dont allons maintenant démontrer l'équivalence sémantique.

Recompilation et validation. La phase de validation démarre par une nouvelle compilation, *sans optimisation*, afin de préserver la structure du code, cette fois en partant du code C émis par le portage initial. D'une façon similaire, nous arrivons à localiser via les informations DWARF le code binaire correspondant à notre code C. Nous sommes à ce stade en possession de deux morceaux de code DBA, l'un venant de cette dernière localisation et l'autre venant de la première compilation. La phase de validation consiste ainsi à *démontrer l'équivalence sémantique des deux codes DBA*.

Pour ce faire, nous utilisons une méthode originale fondée sur un isomorphisme de graphe, facilité par la préservation de la structure du code, et une preuve d'équivalence bloc à bloc, démontrée par solveur SMT. Ceci permet de se débarrasser des boucles dans la traduction logique de la preuve d'équivalence. Cette méthode particulière est utilisable car nous avons la main sur toute la chaîne de compilation et de transformation de portage : ainsi nous faisons en sorte que la structure par blocs de base reste inchangée afin de pouvoir appliquer notre stratégie.

La phase de validation a posteriori de notre traduction permet ainsi à notre technique d'être sûre et par conséquent de s'insérer dans un contexte de vérification sans remettre en cause les propriétés de correction des outils d'analyse. Un autre aspect de cette orientation vers la vérification, concernant une mesure de l'impact pratique sur les outils d'analyse, sera étudié en Sec. 4.2. Revenons tout d'abord sur les détails des transformations qui vont en définitive permettre l'efficacité pratique de la méthode.

3.2 Détails du portage de l'IR vers C

Les détails du portage comportent 9 passes de transformation qui permettent de revenir d'un code assembleur à un code C sémantiquement équivalent. À la suite de ces passes de transformation, le code C est émis.

1. *(Renommage)* La première passe de renommage utilise les informations liées à l'interface, transmise par le DWARF. Ainsi les variables DBA liées aux registres ou zones mémoires sont remplacées par les noms idoines du contexte C. Le DWARF nous permet d'identifier exactement les variables, par exemple pour déterminer si une variable locale est affectée en registre, événement plutôt rare en `x86` compilé sans optimisation, ou dans la pile, comme décalage par rapport au registre (`ebp`).

2. *(Contrôle)* Avant d'entamer des phases de transformations plus intensives, nous contrôlons que les spécifications sont correctes, par exemple qu'il n'y pas

d'entrée non déclarée, pas de sortie non déclarée, ou encore que le code est bien invariant modulo allocation de registres pour ne pas être à la merci du compilateur.

3. *(Déballage)* La seconde étape déballe les sous-éléments accessibles des registres pour pouvoir les substituer directement dans le code DBA. En effet, une affectation d'un registre comme eax se verra décomposée en de multiples affectations restreintes dont celles de ax (les 16 bits de poids faibles d'eax) et al (resp. ah), c'est-à-dire les 8 bits de poids faibles (resp. forts) d'ax. Par la suite ces éléments seront directement substitués en lieu et place des lectures réduites aux mêmes sous-ensembles du registre eax. Cette phase est particulièrement adaptée aux registres xmm, de taille 128 à 512 bits, représentants généralement des vecteurs d'éléments indépendants et plus petits. De manière générale, cette phase permet de gagner en précision sur les registres utilisés pour stocker plusieurs entités indépendantes.

La substitution des sous-éléments, à laquelle succède une phase de filtrage des variables non utilisées permet de propager une information plus précise sur les opérations menées sur les sous-ensembles de registres.

4. *(Conditions de haut niveau)* Les conditionnelles en assembleurs sont des opérations faites sur des indicateurs (*flags*), par exemple de signe, de débordement ou de retenu et non sur une comparaison de haut niveau retournant un booléen. Cette phase réutilise ainsi la technique de Djoudi et coll. [20], fondée sur l´équivalence sémantique démontrée par solveurs SMT, pour recomposer des comparaisons booléennes de haut niveau, plus « lisibles »— qui favorisent également les analyseurs. Comme pour la passe 3, les instructions haut niveau retrouvées sont insérées dans le code DBA, puis une passe de filtrage de variables non utilisées nettoie le code pour ne laisser que les élément nécessaires au calcul de haut niveau.

5. *(Pré-structuration du code)* Nous souhaitons retrouver un code C aussi bien structuré que possible dans le but d'aider les analyseurs (et d'améliorer la lisibilité du code produit). Cette passe recherche ainsi des motifs correspondants aux structures de contrôle C comme les conditionnelles ou les boucles. Pour ce faire, le programme est décomposé en blocs de base, sur lesquels nous allons reconnaître des motifs de haut niveau — comme une conditionnelle avec deux branches et un postlude commun. Ce motif reconnu a lui-même une forme de bloc, sur lequel nous appliquons la même méthode, et ainsi de suite, récursivement. Cette méthode est ainsi appliquée de la base (blocs de base) au sommet pour structurer notre code.

6. *(Mise en forme SSA)* Avant d'appliquer d'autres transformations, le code est mis en forme SSA. Cela permet, comme d'habitude, de faciliter l'implémentation des phases postérieures.

7. *(Propagation de constantes et d'expressions & élimination des sous-expressions communes)* Pour aller plus loin dans la simplification du code, nous procédons ensuite à une phase classique de propagation de constante et d'expressions. Cette deuxième propagation est contrebalancée par une forme d'élimination de sous-

expressions communes globale au morceau d'assembleur en ligne. Ici encore, les éléments nouveaux sont insérés puis le code est filtré.

8. *(Transformation d'opérations bit-à-bit en arithmétique)* Pour faciliter d'une part la lisibilité du code et d'autre part la gestion de celui-ci par certaines analyses, nous transformons autant que faire se peut les opérations bit-à-bit vers des opérations arithmétiques sémantiquement équivalentes. Le code assembleur en ligne comporte fréquemment ce type d'optimisation manuelle, qui peut parfois affecter les analyseurs de code.

9. *(Normalisation de boucle)* Enfin, notre dernière passe normalise la structure des boucles, en particulier les formes de compteur, afin d'en faciliter la gestion par les analyses ultérieures.

4 Expérimentations

Nous évaluons TInA (Sec. 3.1) via deux questions : 1) Est-elle applicable sur le type de code assembleur généralement rencontré dans les logiciels actuels ? 2) Comment se comportent les analyseurs de code lorsque nous portons le code assembleur vers du C ? A-t-on un effet bénéfique sur la qualité d'analyse via l'utilisation de notre méthode ?

Pour aborder ces questions, nous utiliserons trois projets libres qui contiennent un grand nombre de code assembleur en ligne : ALSA [4], pourvoyeur des fonctionnalités son et MIDI au système Linux, ffmpeg [5] le couteau suisse multi-plateforme pour la conversion de son et de vidéo et GMP [6] la bibliothèque GNU de manipulation d'arithmétique en précision arbitraire.

4.1 Généralité de l'applicabilité de la méthode

Pour évaluer l'applicabilité de notre méthode, nous l'employons sur tous les morceaux d'assembleur en ligne trouvé dans les sources des 3 projets suscités, ciblant l'architecture x86. Au total, ces projets contiennent **365** morceaux d'assembleur. Les résultats sont résumés en Table 1.

Nous écartons de l'évaluation les morceaux triviaux (vides ou non utilisés) ou hors-sujet, et rejetons ceux qui violent certaines hypothèses de sûreté (accès non-déclarés par exemple). Un morceau d'assembleur est ici considéré hors-sujet s'il utilise des flottants, non supportés par Binsec, ou des primitives d'accès matériel, non modélisables en C. Le rejet des blocs est principalement issu de la non-conformité des déclarations des entrées/sorties/modifications. Certains morceaux de ffmpeg communiquent ainsi entre eux en séquence par registres xmm sans le déclarer dans leur interface. Nous signalons ceci dans notre prototype comme une erreur et rejetons ces morceaux.

Parmi les morceaux considérés, c'est-à-dire 75% (273/365) des morceaux initiaux, le portage est *réussi et validé à 100%.*

4. `https://www.alsa-project.org/main/index.php/Main_Page`
5. `https://www.ffmpeg.org/`
6. `https://gmplib.org/`

TABLE 1: Applicabilité sur les morceaux asm de 3 projets open-source : ALSA, ffmpeg, GMP

	ALSA	ffmpeg	GMP	TOTAL	
Blocs asm	25	103	237	365	
Triviaux	0	6	13	19	
Hors-sujet	0	19	0	19	
Rejetés	0	55	1	56	
Pertinents	25	25	223	273	
Portés	25	25	223	273	100%
Validés	25	25	223	273	100%
Taille moyenne (instructions)	50	50	7	15	
Taille maximum (instructions)	70	341	31	341	

4.2 Adéquation aux analyses formelles de programme

Nous sélectionnons deux outils représentant des familles distinctes de techniques formelles utilisées dans l'industrie actuellement : l'interprétation abstraite [17,18] et la vérification déductive [23,24]. Nous prenons un représentant par catégorie dans Frama-C [28] : le greffon EVA [10] d'interprétation abstraite et le greffon WP [6] de vérification déductive. Frama-C a un support limité de l'assembleur en ligne fondé sur l'interface du morceau en question, traduit en annotations logiques assigns. Chaque greffon est responsable de la gestion de ces annotations en fonction de sa sémantique sous-jacente.

TABLE 2: Impact du portage sur Frama-C EVA

	ALSA	ffmpeg	GMP	TOTAL	
Fonctions C	22	19	16	57	
Blocs asm	25	26	122	173	
Fonctions avec retour (non-void)	0	9	10	19	
Amélioration de la précision du retour	—	9	1	10	52%
Fonctions avec alarme initiale C	2	9	16	27	
Réduction alarmes C	2	9	13	24	89%
Nouvelles alarmes mémoires asm	12	1	0	13	23%
Impact positif	14	15	13	42	74%

La notion d'*impact positif* englobe ici l'amélioration de la précision, la réduction des alarmes C et l'ajout d'alarmes spécifiques au code porté depuis l'assembleur.

Interprétation abstraite. Le greffon EVA prend en compte les annotations logiques assigns générées, résultant de fait dans une sur-approximation des valeurs possibles pour lesdites variables. Ici, nous montrons comment notre traduction 1) améliore la précision de l'analyse ; 2) impacte les alarmes levées pré- et post-traduction.

Pour ce faire, nous exécutons (abstraitement) **57** fonctions, comprenant 173 blocs assembleur, avec des valeurs abstraites d'entrées (intervalles). Cela nous permet de voir, par rapport à la situation initiale sans portage, si la valeur de retour est plus précise (quand elle existe) ou si certaines alarmes existantes ont pu être désactivées.

La Table 2 résume les résultats obtenus. Nous pouvons constater qu'il y a presque toujours (24/27) réduction d'alarmes post-portage dans le code C commun. Ceci est directement lié à une amélioration générale de la précision de l'analyse puisque les variables modifiées dans le code porté sont maintenant visibles par EVA. Dans la moitié des cas environ (10/19), nous arrivons également à gagner en précision sur le retour de fonction. Au total, 30/33 ($\approx 90\%$) fonctions avec retour ou alarme initiale exhibent une de ces deux améliorations de précision.

Le code C porté contient maintenant lui aussi des alarmes (13/57), auparavant indétectables, qui doivent être prises en compte. Ainsi, nos exemples exhibent d'ailleurs un cas relevant du bogue potentiel dans ffmpeg : le code peut ainsi accéder à l'index -1 du tampon d'entrée. En outre, ce code ne contient aucune documentation ou assertion qui montre que le programmeur avait perçu ce problème : c'est pourquoi nous parlons de bogue potentiel. Sans analyser l'ensemble du programme, nous ne pouvons cependant en déduire qu'il existe un chemin d'exécution qui déclenchera ce bogue potentiel. De manière générale, le code porté met au jour des alarmes additionnelles d'accès mémoire ou de débordement potentiel.

En résumé, sur 74% (42/57) des fonctions au total, nous arrivons à observer un impact positif (plus de précision, extinction des alarmes résultant de l'opacité du code assembleur ou alarmes mémoires dans le code porté) pour la qualité de l'analyse résultante.

Vérification déductive. Pour évaluer notre portage vis-à-vis de la vérification déductive, nous prenons 10 fonctions optimisées en assembleur, dont nous savons vérifier la version C initiale, et le greffon WP [12] de Frama-C. Cela nous permet d'avoir un squelette de preuve sur lequel bâtir la preuve de la version portée. Les 10 exemples détaillés en Table 3 consistent pour moitié d'exemples synthétiques, pour moitié provenant de sources extérieures.

Pour chacune de ces fonctions, nous essayons de prouver les mêmes propriétés fonctionnelles avec deux niveaux d'optimisation : le premier est *naïf* (passes 1, 2, 5 et 6 du portage vers C détaillé en Sec. 3.2), le second, TInA, comprend toutes les passes (1 à 9). Pour référence, nous essayons aussi de démontrer le programme sans portage — dans ce cas le contenu du morceau d'assembleur est invisible. Les résultats sont résumés en Table 4. TInA, comparé à un portage naïf, permet à la fois de réduire le nombre d'obligations de preuve car le code comporte moins d'instructions, et de plus haut niveau, et de faire démontrer lesdites obligations (*100% de réussite* sur les 10 exemples). Ces résultats démontrent que les passes

Table 3: Détail des fonctions testées pour Frama-C WP

	Fonction	Description	Origine
Calculs	saturated_sub	Maximum entre 0 et la soustraction entière (« grands entiers ») des 2 entrées	—
	saturated_add	Minimum entre MAX_UINT et l'addition entière (« grands entiers ») des 2 entrées	—
	log2	Plus grande puissance de 2 d'un entier	—
	mid_pred	Valeur médiane entre 3 entrées	ffmpeg
Tableaux	memset	Remplit le contenu du tableau via l'entrée	—
	count	Compte le nombre d'occurrences des entrées dans le tableau	ACSL by example
	max_element	Premier index du plus grand élément du tableau	ACSL by example
	sum_array	Somme des éléments du tableau	ACSL by example
Chaînes de caractères	streq	Teste l'égalité de deux chaînes de caractères	—
	strnlen	Calcule la taille de la chaîne (ou du tampon en l'absence de '\0')	fast strlen

Table 4: Impact du portage sur Frama-C WP

		Portage					
		Sans	Naïf		TInA	
	Instructions	OPs[a]	OPs	Invs.[b]	OPs	Invs.
saturated_sub	2	✗ 3 / 6	✓ **29 / 29**	0	✓ **7 / 7**	0
saturated_add	2	✗ 3 / 6	✗ 27 / 29	0	✓ **7 / 7**	0
log2	1	✗ 2 / 4	✗ 24 / 28	5	✓ **16 / 16**	5
mid_pred	7	✗ 3 / 12	✗ 61 / 68	0	✓ **19 / 19**	0
memset	9	✗ 1 / 3	✗ 38 / 44	0	✓ **19 / 19**	0
count	8	✗ 3 / 5	✗ 56 / 61	4	✓ **17 / 17**	4
max_element	10	✗ 3 / 7	✗ 59 / 69	7	✓ **33 / 33**	7
sum_array	20	✗ 3 / 5	✗ 333 / 336	7	✓ **21 / 21**	7
streq	10	✗ 3 / 6	✗ 139 / 146	6	✓ **18 / 18**	6
strnlen	16	✗ 3 / 8	✗ 69 / 79	6	✓ **27 / 27**	6

[a] Obligations de Preuve [b] Invariants

supplémentaires implémentées, potentiellement superflues concernant le critère de correction, sont en réalité nécessaires pour permettre l'intégration dans un processus de vérification.

4.3 Conclusion

Les expériences menées établissent que la technique proposée permet de traiter le code assembleur en ligne dans un contexte C. Ainsi, tous les codes assembleur en ligne à la portée de notre outil sont portés et validés (Sec. 4.1).

Par ailleurs, le code porté exhibe un bon degré de *vérifiabilité* (Sec. 4.2). Pour l'analyse par interprétation abstraite (EVA), le portage a un impact positif, soit en réduisant le nombre d'alarmes du code C commun, soit en améliorant la précision, soit encore en mettant au jour de nouvelles alarmes mémoire du code porté, qui sont parfois des bogues potentiels du programme. Le cas de la vérification déductive (WP) montre que le portage a de plus besoin de passes de transformations pour permettre de totalement vérifier en pratique nos exemples.

5 Travaux connexes

Portage de code assembleur et vérification. Maus [30,**?**] propose une méthode générique qui simule le comportement des instructions assembleur dans une machine virtuelle écrite en C. Ces travaux proviennent du projet Verisoft pour vérifier le code d'un hyperviseur qui contient du code mixte bas niveau. La technique de Maus utilise VCC [15] pour écrire et démontrer les conditions de vérification de l'état de la machine. Contrairement à notre approche, le code virtuel contient tous les détails (par exemple les indicateurs) du code bas niveau alors que nous nous efforçons d'obtenir un code le plus haut niveau possible.

Des travaux successifs de Schmaltz et Shadrin [36] visent à prouver l'adéquation au niveau de l'interface binaire-programm (*ABI*) de la partie assembleur. Cette méthode est cependant circonscrite à l'assembleur MASM (Microsoft Macro Assembler) et Windows. Notre méthode, bien qu'ici appliquée à l'assembleur en ligne de GCC (et clang), est effectivement indépendante du dialecte d'assembleur, car elle fonctionne à partir d'analyseurs de code binaire applicables à un plus grand nombre d'architectures.

Fehnker et coll. [22] traitent l'assembleur en ligne pour l'architecture ARM à travers l'usage de *model checking* pour analyser du code mixte. Cette solution est cependant purement syntaxique : ainsi, elle se limite à un type de dialecte assembleur pour une architecture, mais elle perd aussi la correction que nous visons. Cette perte peut être un compromis adéquat, mais pas dans le cadre d'analyses formelles correctes comme l'interprétation abstraite.

Corteggiani et coll. [16] utilise du portage de code. Cependant, leur but est de faire des analyses symboliques dynamiques sur le code porté — sans viser d'autres analyses. Or, certaines analyses vont fonctionner sans aucune technique avancée de traduction alors que d'autres vont nécessiter plus d'effort. Nous avons vu par exemple que le greffon WP de Frama-C nécessitait certaines

passes d'optimisations pour être utilisé tel quel. Les analyses de type exécution symbolique sont a priori celles qui ont le moins besoin de transformation pour fonctionner. Par ailleurs, la correction de la démarche n'est pas abordée.

Décompilation. Cette technique [32] a pour but de recouvrer le code source d'origine (ou un très proche) d'un code exécutable. Cet objectif, très difficile, nécessite de rechercher l'information perdue durant la compilation [13,32]. En dépit de progrès récents [9], la décompilation reste un défi scientifique ouvert. Cependant, cela permet tout de même d'améliorer la compréhension d'un programme, par exemple en rétro-ingénierie. Ce but n'est pas toujours en accord avec celui que nous avons de nous insérer dans un processus de vérification — par exemple ce n'est pas forcément nécessaire de produire un code source en tout point valide.

Schulte et coll. [37] utilisent pour garantir la correction une technique *search-based* afin de produire un code source qui se compile en code binaire égal octet pour octet au code binaire d'origine. Cette technique, lorsqu'elle termine, assure par construction la correction mais elle ne s'applique pour l'instant en pratique qu'à de petits exemples, avec un succès mitigé.

Nous réutilisons également des technique de rétro-ingénierie [29,**?**] permettant d'inférer le type des registres et zones mémoires via la façon dont ils sont utilisés. Ceci nous aide à renforcer notre « système de types » bien que nous ne construisions aucun type non dérivé de celui des entrées, et donc connu par l'interface.

En décompilation, un grand défi consiste à récupérer la liste des instructions et le graphe de flot de contrôle du code [1,31]. Bien que ce problème soit très difficile, notamment dans le cas de code offensif de type malware, le code usuel a une bien plus grande régularité, avec un nombre conséquent de motifs, permettant en pratique une bonne reconstruction en utilisant des méthodes incorrectes — donc sans garantie. Notre cas est encore plus favorable : les morceaux d'assembleur sont en général encore plus limités au niveau structurel (flot de contrôle simple, pas de saut calculé dynamiquement) et nous avons accès à l'ensemble de la chaîne de compilation.

Analyse de programme au niveau binaire. Depuis plus d'une décennie maintenant, la communauté d'analyse de programme a consacré d'importants efforts pour traiter directement le code exécutable [2] soit pour pouvoir analyser les codes dont le source n'est pas disponible (composants sur étagère, code hérité, malware), soit pour vérifier le code tel qu'il s'exécute véritablement. Ces efforts se sont concentrés sur le recouvrement d'abstractions de haut niveau [4,20,**?**,**?**,**?**] et le calcul d'invariants.

En outre, plusieurs « porteurs » (*lifters*) ont été proposés récemment, réduisant ainsi les divers jeux d'instructions à un petit nombre de primitives bien définies d'un langage intermédiaire. Si ceux-ci sont éprouvés en pratique [26], nous pourrions en dériver encore plus de confiance s'ils étaient générés automatiquement, par exemple à partir de spécifications similaires à celles, récentes, de ARM [34].

Validation de traduction et équivalence de code. Pour garantir la sûreté de notre portage, nous nous inscrivons dans la mouvance de la validation de la traduction [33,**?**,**?**], technique utilisée par exemple pour l'allocation de registre de CompCert [7]. Ceci nous permet à moindre coût de faire reposer nos besoins formels de correction sur des outils disponibles et éprouvés (en l'occurrence les solveurs SMT), utilisables en boîte noire, plutôt de nous atteler à une démonstration formelle complète de la correction de l'ensemble de la chaîne.

6 Conclusion

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

Références

1. D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 583–600. USENIX Association, 2016.
2. G. Balakrishnan and T. W. Reps. WYSINWYX : what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6) :23 :1–23 :84, 2010.
3. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In G. Gopalakrishnan and S. 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.
4. S. Bardin, P. Herrmann, and F. Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In R. Jhala and D. 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.
5. C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.
6. P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Manual*, Frama-C Chlorine-20180501 edition, 2018.
7. S. Blazy, B. Robillard, and A. W. Appel. Formal Verification of Coalescing Graph-Coloring Register Allocation. In A. D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010.

8. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. *BAP : A Binary Analysis Platform* , pages 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
9. D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In S. T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 353–368. USENIX Association, 2013.
10. D. 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.
11. C. Cadar, D. Dunbar, and D. R. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In R. Draves and R. 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.
12. N. Carvalho, C. da Silva Sousa, J. S. Pinto, and A. Tomb. Formal Verification of kLIBC with the WP Frama-C Plug-in. In J. M. Badger and K. Y. 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.
13. B.-Y. E. Chang, M. Harren, and G. C. Necula. *Analysis of Low-Level Code Using Cooperating Decompilers*, pages 318–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
14. C. Cifuentes. Interprocedural data flow decompilation. *J. Prog. Lang.*, 4(2) :77–99, 1996.
15. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. *VCC : A Practical System for Verifying Concurrent C*, pages 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
16. N. Corteggiani, G. Camurati, and A. Francillon. Inception : System-Wide Security Testing of Real-World Embedded Systems Software. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 309–326. USENIX Association, 2018.
17. P. Cousot and R. Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In R. M. Graham, M. A. Harrison, and R. 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.
18. P. Cousot and R. Cousot. Abstract interpretation : past, present and future. In T. A. Henzinger and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 2 :1–2 :10. ACM, 2014.
19. A. Djoudi and S. Bardin. *BINSEC : Binary Code Analysis with Low-Level Regions* , pages 212–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
20. A. Djoudi, S. Bardin, and É. Goubault. *Recovering High-Level Conditions from Binary Programs* , pages 235–253. Springer International Publishing, Cham, 2016.

21. DWARF Debugging Information Format Committee. *DWARF Debugging Information Format 5*, 2017.
22. 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.
23. 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.
24. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10) :576–580, 1969.
25. Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2016.
26. S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In G. Rosu, M. D. Penta, and T. 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.
27. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7) :385–394, July 1976.
28. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c : A software analysis perspective. *Formal Asp. Comput.*, 27(3) :573–609, 2015.
29. J. Lee, T. Avgerinos, and D. Brumley. TIE : Principled Reverse Engineering of Types in Binary Programs. In *NDSS*, 2011.
30. S. Maus, M. Moskal, and W. Schulte. *Vx86 : x86 Assembler Simulated in C Powered by Automated Theorem Proving*, pages 284–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
31. X. Meng and B. P. Miller. Binary code is not easy. In A. Zeller and A. 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.
32. 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.
33. G. C. Necula. Translation validation for an optimizing compiler. In M. 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.
34. A. Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In R. Piskac and M. Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 161–168. IEEE, 2016.
35. M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. 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 2018, Williamsburg, VA, USA, March 25-25, 2018*, pages 84–99. ACM, 2018.

36. S. Schmaltz and A. Shadrin. *Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT*, pages 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

37. E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Logino. Evolving Exact Decompilation. In *BAR 2018, Workshop on Binary Analysis Research, San Diego, California, USA, February 18, 2018*, 2018.