# Un pas de plus vers le passage de pointeurs en compilation sécurisée

Florian Groult

25 septembre 2018

# Table des matières

# Remerciements

D'un point de vue personnel, je remercie du fond du cœur ma chère et tendre, ma fiancée et future mère de mon fils pour sa patience et son soutien constant durant le stage et les nuits où j'ai dû rester sur Paris, et pour sa relecture, sa critique et son écoute. Je remercie aussi ma famille entière (dont la belle famille) pour tout, tout simplement.

Je tiens à remercier tous mes collègues et amis de l'équipe Prosecco pour leur accueil très chaleureux durant ce stage et pour le contrat d'Ingénieur de Recherche (IGR) à venir. Tout particulièrement, je tiens à remercier Cătălin d'avoir cru en moi et de m'avoir pris en tant que stagiaire et IGR et Rob pour son aide régulière et attentive. Je remercie également toutes ces personnes qui ont fait de ces 6 mois de stage ma meilleure expérience professionnelle et pédagogique, Cătălin et Rob à nouveau, mais aussi (sans ordre particulier) Carmine, Jérémy, Aaron, Victor, Blipp, Benjamin, Denis, Prasad, Éric, Kenji, Nadim, Bruno, Iness, Natasha, Marina, Karthik, Bruno, Marc, Danel, Théo, Amal, Harry, mais aussi Guido que je viens à peine de rencontrer avant que je ne m'isole du reste du monde pour finir mon rapport de stage, et Arthur qui nous a visité bien trop tôt durant mon stage pour que je puisse vraiment apprendre SSReflect de sa part ;-)

Je remercie aussi Frédéric Loulergue pour son intérêt envers mon parcours et mes travaux ainsi que sa proposition inopinée que j'ai fortement hésité à accepter.

Merci également à Vivien Pelletier d'avoir suivi mon stage et d'être passé à l'Inria.

Je remercie à nouveau Denis et Prasad pour leurs conseils professionnels et de m'avoir aidé à m'y retrouver.

Je remercie l'Inria et particulièrement Cătălin et nos assistants Mathieu et Anna de m'avoir supporté dans toutes mes démarches très aimablement, et de m'avoir donné l'occasion de pouvoir étudier à l'école d'été FOSAD en août. Je remercie à nouveau ceux avec qui j'ai passé cette semaine géniale en Italie et spécialement Carmine pour nous avoir fait découvrir ce pays qui est le sien.

Je remercie aussi l'équipe Gallium pour ses séminaires réguliers et enrichissants, l'équipe Secret pour leur seul séminaire auquel j'ai pu assister et notre équipe pour les siens également.

Un grand merci également à l'IUT et au Département informatique de l'Université d'Orléans pour mon éducation et pendant laquelle j'ai eu certains des meilleurs enseignants durant tout mon cursus scolaire.

Merci à Benjamin, Denis et Carmine pour les chouquettes régulières !

Merci à ceux à qui je dois moultes boissons : Victor, Carmine et Jéremy.

Et merci à vous, lecteur, de lire le fruit de mon travail.

# Introduction

La sécurité informatique est une perpétuelle poursuite entre la découverte de vecteurs d'attaques avec leurs failles exploitables et les contre-mesures pour y remédier. Implémentez une librairie cryptographique en optimisant le temps d'exécution au maximum et admirez les attaquants dérober la clé en quelques attaques temporelles ! Comblez cette faille sans attendre mais faites toujours usage du cache comme auparavant, l'attaquant se met à refaire une attaque temporelle, mais en effaçant quelques lignes du cache ! Résolvez ce problème et voici que votre attaquant change les bits à l'aide d'un laser ! Rajoutez des instructions idempotentes pour rendre beaucoup plus compliqué cette tâche et vous vous rendez compte problème sous-jacent est en fait facilement solvable par un ordinateur quantique ! !

Bien que nous n'en soyons pas vraiment à cette dernière étape cela montre bien la diversité des attaques possibles et la difficulté de protéger un secret dans un programme. Heureusement, pour faire face à cela, des méthodes formelles ont été créées pour pouvoir modéliser et prouver (jusqu'à un certain point) des algorithmes respectant certaines propriétés C'est la démarche employée dans plusieurs projets où la sûreté est critique tels que l'aéro(-nautique et -spatiale), les transports de façon plus générales ou les librairies cryptographiques. Il est possible d'avoir un logiciel respectant vos impératifs de sûreté. Maintenant, vous avez une garantie que votre fusée n'explosera plus en plein décollage à cause d'erreurs informatiques.

Néanmoins… ce genre de démarche ne marche que lorsque TOUT le logiciel est passé par ce procédé ! Embarquez une librairie cryptographique finement étudiée et "à toute épreuve" (BoringSSL[27] ou HACL*[65] à tout hasard) dans une application ou les contributeurs se comptent par centaines, les commits par milliers et les lignes de codes par millions (Chromium ou Firefox), et vous perdez toutes les garanties de sûreté. La librairie cryptographique peut bien être "infaillible"[1], aucun exploit n'en émanera. Mais une petite faille bien exploitée dans le reste du logiciel et il sera peut être possible pour l'attaquant d'atteindre une portion du code sensible avec un saut ou de lire la clé en mémoire…[2]

C'est ce dernier point que le projet SECOMP tente de résoudre : préserver les garanties prouvées même lorsque cette partie du logiciel est en lien avec du code vulnérable. Pour cela, ce projet explore deux versants :

- Formaliser précisément ce que l'on attend de la compilation sécurisée, et quelles preuves, d'un point de vue haut niveau, permettent de les obtenir ;
- Fournir une preuve de concept de compilation sécurisée en utilisant l'une des techniques de preuves de haut niveau "facile" sur un langage impératif non-sûr[3] simple, avec *buffers* et procédures mais Turing-complet ; et tenter de rapprocher de plus en plus ce langage du C.

Mes travaux effectués pendant ce stage ont porté sur le second versant de ce projet. Pour vous présenter le projet et l'accomplissement de ce stage, je vous présenterai :

- Dans un premier temps, les différentes notions nécessaires à la compréhension de ce compilateur[4],
- Ensuite je vous familiariserai avec le projet et son contexte,
- Et enfin je vous ferai part du contenu à proprement parler de mon stage, en vous donnant :
  - Les choix abordés pendant la conception "papier" et
  - Les différents problèmes rencontrés.

Bien entendu, nous verrons les différentes pistes abordables à l'issue de ces travaux.

1. évidemment, nous ne sommes pas dans un cadre avec des ordinateurs quantiques pour nous simplifier l'explication
2. Bien évidemment, ce genre de logiciel essaie de rendre ces failles les plus inaccessibles possible. Cela reste normalement un défi de réaliser l'un de ces exploits sus-mentionnés.
3. C'est-à-dire ne garantissant pas la sûreté mémoire[7]
4. Nous partons du principe que le lecteur possède le niveau de connaissance d'un étudiant ayant fait un cursus M2 Informatique Nomade, Intelligence et Sécurité (INIS), parcous Sécurité et Sûreté du Logicien (SSL).

# Notions préalables et état de l'art

Vous l'aurez deviné, nous allons parler en long, en large et en travers de langage de programmation, de compilateurs, de sémantique et j'en passe. Pour une bonne compréhension du sujet, il est important de définir de nombreux points, à commencer par ce que l'on va entendre par langage de programmation, où la notion est légèrement plus poussée que celle vue durant le Master.

Tout d'abord qu'entend-on par langage de programmation ? D'un point de vue formel, il se cache deux concepts sous l'ombrelle de "langage de programmation" : la syntaxe de celui-ci –c'est à dire son écriture correcte– et sa sémantique –son sens–.

La syntaxe est un point qui a été vu en détail durant le cycle Licence-Master d'informatique à l'Université d'Orléans, c'est pourquoi nous ne nous attarderons pas là-dessus. Néanmoins, la sémantique, bien qu'abordée dans le cours de compilation, mérite d'être plus détaillée.

## 3.1 Sémantique informelle et *undefined behavior*

Lorsque l'on parle de sémantique, cela signifie associer à un programme un sens. Le plus souvent, dans les standards de langages ou la documentation d'un compilateur, c'est une sémantique informelle qui est employée. On y décrira l'effet de telle construction du langage de programmation[1] selon un langage naturel.

Ce type de description, bien que très compréhensible par n'importe quel programmeur, est le plus souvent incomplète et ne permet pas de se rendre compte des cas extrêmes (ou *corner cases*).

De plus, ces descriptions peuvent être très évasives sur certains détails. Le langage C, quelque soit sa définition selon les différents standards, en est un exemple frappant. Dans la dernière version publique du standard C11[18, 34],

- 120 comportements sont marqués comme laissés à charge dans l'implémentation (*Implementation-defined behavior*), c'est à dire que le comportement doit être documenté et consistant,
- 58 sont marqués non-spécifiés (*unspecified behavior*), un comportement analogue à *Implementation-defined* mais dont la documentation n'est pas obligatoire,
- Et **203** (‼) sont marqués comme non-définis (*undefined behavior*) ! Un comportement indéfini signifie qu'un tel programme n'a pas de sens selon le standard et que tout peut arriver. N'importe quel comportement que le compilateur décide de lui donner –que ce soit une fonction non-déterministe, nettoyer intégralement le disque dur ou envoyer tous fichiers de l'ordinateur sur un espace de partage public– est valide selon le standard.

Écrire un programme sans *undefined behavior* est une tâche difficile et bien que les compilateurs vont adopter un comportement raisonnable dans ces cas, il est bien souvent possible qu'une vulnérabilité existe lorsqu'un programme exhibe un tel comportement, comme lire/écrire en dehors de sa mémoire allouée.

Tous ces comportements sont explicitement non-définis pour des raisons de performance et permettre au compilateur d'optimiser au plus les programmes générés[57, 45], en ne se souciant plus des *corner cases*, mais nous ne nous attarderons pas plus sur cette notion.

1. On peut parler plus formellement d'élément syntaxique, un élément de l'arbre de syntaxe abstrait (AST) qui est donc extrait d'un programme syntaxiquement correct.

## 3.2 Sémantiques formelles[16, 21, 22]

À cette sémantique informelle, on oppose naturellement des sémantiques formelles qui associent aux différentes constructions du langage un objet mathématique plutôt qu'un sens intuitif. Trois grandes familles de sémantiques existent et selon celle choisie, différentes logiques ou structures mathématiques sont utilisées :

- Avec la sémantique opérationnelle l'exécution du programme est modélisée en tant que transition entre états,
- En utilisant la sémantique dénotationelle, le programme est associé à une fonction mathématique,
- À l'aide de la sémantique axiomatique, les programmes sont annotés de prédicats.

### Notations

Durant la suite du rapport, nous allons utiliser des couleurs pour différentier les programmes, nous allons utiliser des polices :

- sans sérif bleue pour les éléments au niveau source,
- sérifée grasse **rouge** pour ceux au niveau **cible**,
- noire comme ceci ou $cela$ pour les éléments communs aux deux langages.

De plus, nous utiliserons les notations suivantes :

- p↓ pour parler d'un programme (dans le sens général) p compilé,
- par convention, P pour un programme, complet ou partiel et
- C pour un contexte arbitraire, considéré comme un ensemble de composants antagonistes,
- C [P] ou C ∪ P pour un programme P lié (*linked*) à un contexte arbitraire C
- Et enfin la relation p $\rightsquigarrow$ t pour exprimer qu'un programme c émet une trace t à l'exécution.

### 3.2.1 Sémantique opérationnelle

La sémantique opérationnelle va donner un sens au programme lors du parcours de l'*AST* : à chaque élément syntaxique de celui-ci l'on va tenter de faire correspondre une des règles de la sémantique du langage, règles ayant la forme d'une implication logique et appelées règles d'inférence.

Ces règles d'inférences sont constituées d'une relation, ou *jugement d'évaluation*, de la forme $\sigma \vdash e \Rightarrow v$ –signifiant grossièrement que dans l'environnement ou état $\sigma$, l'expression $e$ s'évalue en la valeur $v$–, ainsi que d'hypothèses à cette relation.

Dans un langage fonctionnel (pur) on obtient effectivement une valeur à l'issue de cette règle. Pour modéliser tout l'aspect impur dans un langage impératif, on va souvent considérer que la valeur $e$ va être un couple (observation sémantique, état final) et l'expression $v$ permet donc de faire la transition entre l'état initial et l'état final en relevant cette observation. Cette dernière peut être, par exemple, un évènement observable en dehors du programme comme un évènement d'entrée-sortie ou un appel d'une fonction en dehors du programme.

C'est pourquoi, le plus souvent, on va associer à cette sémantique un système de transitions d'états, une notion englobant les automates finis. Évaluer la sémantique opérationnelle d'un programme revient en quelque sorte à l'"interpréter". De ce fait, donner la sémantique formelle d'un langage donne un interpréteur de référence pour celui-ci.

Cette sémantique est le plus souvent utilisée de prouver le respect de certaines propriétés du langage (sûreté de type, sûreté mémoire, etc.) dans son ensemble car les règles d'inférences ne dépendent que du langage.

Cette famille de sémantique est principalement classée en deux catégories :

- Les sémantiques opérationnelles à grand pas ou naturelles (*big-step semantics*) où les transitions empruntées englobent des constructions comme les conditionnelles ou les boucles en intégrité,
- Les sémantiques opérationnelles à petit pas ou structurées (*small-step semantics*) dont les transitions vont être les plus petites possibles, comprenant chaque évaluation de variable.

Les sémantiques à grand pas ont l'avantage d'être plus simples et donc souvent utilisées pour modéliser des langages complexes. Néanmoins elles ne permettent pas –dans leur forme initiale– de modéliser des programmes dont l'exécution ne termine pas[2].

### 3.2.2 Sémantique dénotationnelle

La sémantique dénotationnelle permet de construire des objets mathématiques appelés dénotations décrivant le sens des expressions/déclarations du langage. D'une façon similaire à la précédente, les différentes dénotations sont déduites par la syntaxe du programme, mais les étapes intermédiaires (détails de l'exécution) sont ignorés : on ne conserve pas l'état du programme.

Les notions utilisées sont proches de celles que l'on peut voir en interprétation abstraite. Ces fonctions ont pour ensembles d'application et de définitions des ordres partiels complets (un sous-ensemble des treillis complets).

Nous n'allons pas détailler outre mesure cette notion que je n'ai pas eu l'occasion de voir durant mon stage.

### 3.2.3 Sémantique axiomatique

La sémantique axiomatique va associer à des portions de programme –fonctions ou programmes entiers principalement, mais aussi expressions/*statements* – des assertions logiques (ou prédicats) devant bien entendu être respectées. La principale instance de cette sémantique, la logique de *Floyd-Hoare*, va annoter un programme $S$ (pour *Statement*) avec les prédicats $p$ et $q$, où $p$ est une pré-condition et $q$ une post-condition, sous la forme d'un triple (de Hoare) $pSq$. Ce dernier signifie que si $p$ est vrai avant que $S$ soit exécuté, et si $S$ termine, alors $q$ est vrai. Comme dans la sémantique opérationnelle, ce triple sert de jugement d'évaluation et est utilisé dans des règles d'inférences : un triple $pSq$ est vrai si il respecte certaines hypothèses.

Si l'on fournit un langage d'une sémantique axiomatique, on a alors un ensemble de règles sur les différentes constructions syntaxiques du langage. Mais contrairement à la sémantique opérationnelle, seule la valeur de vérité de ces règles a de l'importance (on ne cherche pas à connaître l'état intégral du programme à chaque étape) et certaines de ces règles sont des axiomes (dont on a pas à prouver les prémisses). Ainsi, lorsque l'on annote un programme de pré- et post-conditions, il faut en vérifier les différentes hypothèses qui peuvent être aussi des triples, jusqu'à ce que la preuve finisse.

Cette logique et celles qui en découlent (les différentes logiques de séparation[59]) sont très utilisées pour la preuve de programmes et plus particulièrement de programmes manipulant le tas.

Il existe de nombreuses extensions et ramifications de ces sémantiques, mais là n'est pas le sujet de ce rapport.

Le projet dans lequel je suis impliqué comprend quatre langages : un source, un intermédiaire et deux langages. Ils ont des sémantiques opérationnelles à petits pas, mais aussi aux niveaux source et intermédiaire, des sémantiques de traces, une sémantique un peu à part que nous allons voir ici.

## 3.3 Traces et propriétés[21]exploringRobust

### 3.3.1 Modèles de trace

Une trace représente une exécution d'un programme. Sa définition dépend de la précision requise pour cette représentation. Classiquement, dans l'analyse des systèmes réactifs (serveur, système d'exploitation, etc.) et dans les domaines de l'analyse statique formelle (comme l'interprétation abstraite), elle est constituée d'une suite d'états[43]. Mais *CompCert* a également introduit un modèle de traces à base d'évènements (observables en dehors du programme[48] ) enrichis d'informations comme les arguments de fonctions ou valeurs de retour.

Ce modèle a été choisi pour avoir plus de souplesse. En effet, *CompCert* vise à conserver la sémantique d'un programme entre les langages source (C) et cibles (les assembleurs x86, ARM, PowerPC, RISC-V) et donc les traces émises. Dans le cas d'un compilateur optimisant (*optimizing compiler*), une telle représentation a été choisie car utiliser une suite d'états aurait obligé à conserver l'ordre de ceux-ci lors des différents niveaux. Si c'était le cas, la plupart des optimisations ne seraient pas possibles car le compilateur ne pourrait réordonner le code. Utiliser un modèle à base d'évènements observables permet d'effectuer ce réarrangement tant qu'il permet au programme d'émettre la même suite d'évènements. De nombreux compilateurs certifiés ont aussi pris ce modèle de trace[42, 52, 60, 10, 61, 5].

Par ailleurs, une trace peut être infinie, ce qui permet de modéliser l'exécution d'un programme réactif même avec le modèle de traces à base d'évènements.

2. A l'aide d'un raisonnement co-inductif, il est possible de modéliser des programmes divergents, voir présentation de *M. Xavier Leroy*[47, vidéo à 34 :45, 50, slide 169 (177 en réalité)] pour un aperçu des raisons à utiliser l'une ou l'autre de ces sémantiques.

C'est ce modèle qui a été choisi et enrichi pour les projets de *SECOMP* mais avec quelques différences de nomenclatures dans [1]. Dans ce dernier article, une trace (appelée comportement, ou *behavior* dans *CompCert*) donne le résultat d'une seule exécution du programme et un comportement/*behavior* d'un programme est l'ensemble de ses traces. Dans la suite du rapport, nous allons plutôt utiliser la notation de *CompCert*.

Pour revenir sur cette notion de résultat d'une exécution du programme, ou comportement, cela signifie que le programme peut :

- Terminer avec une trace finie[3],
- Réagir indéfiniment / diverger de façon observable, c'est à dire avec une trace infinie,
- Diverger silencieusement c'est-à-dire quand il y a non terminaison du programme mais aucun évènement n'est émis à partir d'un certain moment et
- Finir à cause d'une erreur.

### 3.3.2 Hyper-propriétés et propriétés de trace(s)

Après avoir vu ce qu'est une trace, on pourrait se demander quelle est son utilité. Ces modélisations d'exécution de programme peuvent servir à vérifier certaines propriétés, appartenant à différentes classes selon leur "complexité", dans une hiérarchie définie relativement récemment.

D'une façon assez convenue, on voudrait pouvoir exprimer le fait qu'un programme ne rencontre pas d'erreur, diverge, termine, exécute une action (vivacité ou *liveness*), n'exécute jamais une action (sûreté ou *safety*), ne laisse pas un secret fuiter, etc. Différentes logiques temporelles expriment cela, comme CTL[4] ou LTL[5], mais les traces et états d'un programme le permettent aussi.

Ces différentes propriétés sont des ensembles et vérifier qu'un programme respecte une de ces propriétés signifie que l'ensemble des états, des traces voire des ensembles de traces de ce programme sont inclus dans cette propriété.

Pour donner explicitement cette hiérarchie où chaque classe est incluse dans celle d'après, on a :

- Les propriétés d'états,
- Les propriétés de traces,
- Les hyperpropriétés (de traces) et
- Les hyperpropriétés relationnelles (de traces).

#### 3.3.2.1 Les propriétés d'états

Les propriétés d'états sont des ensembles d'états. La divergence d'un programme ou l'absence d'erreurs en font partie.

#### 3.3.2.2 Les propriétés de traces

Les propriétés de traces sont des ensembles de traces. Un programme respecte une telle propriété si toutes ses exécutions y appartiennent.

On peut modéliser la terminaison d'un programme ou une propriété fonctionnelle ("le programme peut être modélisé par une fonction et en voici la définition…") par des propriétés de traces.

Parmi ces propriétés de trace, il est intéressant de distinguer deux grandes familles : les propriétés de sûreté (*safety properties*) et celles de vivacité (*liveness/dense properties*).

##### 3.3.2.2.1 Les propriétés de sûreté

Les propriétés de sûreté permettent d'exprimer qu'un évènement (indésirable) n'arrivera pas. Ces propriétés incluent également les propriétés d'états. Une telle propriété peut être réfutée par une trace finie. En effet, il suffit de montrer une exécution où cet évènement arrive pour que cette propriété ne tienne plus.

3. Et, uniquement dans *CompCert*, une valeur de retour, le langage manipulé, *Clight*, étant à base d'expressions.
4. Logique Temporelle Arborescente ou *Computational Tree Logic*
5. Logique Temporelle Linéraire ou *Linear Tree Logic*

#### 3.3.2.2.2 Les propriétés de vivacité

Les propriétés de vivacité, à l'opposé, permettent d'exprimer qu'un évènement (voulu) arrive éventuellement. Cette famille ne contient pas les propriétés d'états. De façon complémentaire à celles de sûreté, pour montrer qu'une propriété de vivacité est fausse, il suffit de donner une trace cette fois infinie. Effectivement, montrer qu'un évènement n'arrive jamais dans un programme réactif revient à montrer une exécution intégrale et donc divergente[6]. Cette différence peut être plus problématique lorsque l'on veut prouver ce type de propriété de trace.

Chaque propriété de trace est en fait composé d'une partie vivacité et d'une partie sûreté (qui évidemment peuvent être nulles).

### 3.3.2.3 Les hyperpropriétés (de trace)[20]

Les hyperpropriétés (de trace) sont des ensembles d'ensembles de traces.

Prenons l'exemple canonique de la noninterférence qui aide à mieux comprendre celles-ci. Intuitivement, la plus connue des notions d'interférence tient si, pour un programme dont on a partitionné les entrées et sorties selon des priorités voulues (haute et basse), les sorties de haute priorité ne sont déterminées que par les entrées de haute priorité, de façon analogue au fait qu'une fonction est déterministe. C'est à dire que sur toutes les exécutions de ce programme, si il reçoit les mêmes entrées de haute priorité, quelles que soient les entrées de basse priorité, celui-ci renverra les mêmes sorties de haute priorité.

Pour cette propriété, contrairement à celles de traces, il ne suffit plus de vérifier que **toutes les exécutions** du programme appartiennent à un ensemble. mais il faut vérifier **tous les ensembles d'exécutions**. Plus précisément les ensembles pour lesquels les entrées de priorité haute sont fixées, et donc toutes les exécutions avec les entrées de priorité basse varient.

Pour revenir au cas général des hyperpropriétés, un programme respecte une de celles-ci si les ensembles d'ensembles de traces appartient à cette propriété.

De façon analogue aux propriétés de trace, l'ensemble des hyperpropriétés est découpé selon les ensembles d'hypersûretés (*hypersafety*) et d'hypervivacités (*hyperliveness*). Les techniques de preuves pour réfuter de telles hyperpropriétés sont analogues à celles pour les propriétés de sûreté et de vivacité.

Ainsi, la notion de noninterférence que nous avons vu plus haut est une hypersûreté et peut être réfutée en donnant deux exécutions dont les entrées de haute priorité sont les mêmes et les entrées et sorties de basse priorité diffèrent.

### 3.3.2.4 Les hyperpropriétés relationnelles (de trace)

Finalement, une nouveauté du projet : Les hyperpropriétés relationnelles (de trace) sont des hyperpropriétés sur différents programmes ! C'est une famille de propriété comprenant l'équivalence de trace entre différents programmes, ou mieux connue sous le nom de *full abstraction*. C'est une propriété très importante puisque pendant longtemps considérée comme la propriété à atteindre dans les différents projets visant à réaliser de la compilation sécurisée. Nous allons en parler plus en détails par la suite5.1.2.

Maintenant que nous avons bien défini les différentes notions de sémantique, trace et propriétés que l'on utilisera par la suite, il nous reste à définir comment vérifier celles-ci.

## 3.4 Méthodes formelles

### 3.4.1 Vue d'ensemble

Pour fournir une preuve qu'un programme respecte une propriété, plusieurs choix s'offrent. La réalité étant dure, le théorème de *Rice* nous rappelle à l'ordre en déclarant qu'aucun algorithme ne permet de vérifier n'importe quelle propriété non-triviale pour un programme arbitraire (avec un modèle de calcul analogue à une machine de Turing). C'est pourquoi d'autres méthodes (formelles bien entendu) sont nécessaires pour prouver le respect d'une propriété :

- Considérer une approximation sûre du programme, une démarche abordée dans diverses analyses formelles (statiques ou dynamiques) telles que l'interprétation abstraite, l'exécution symbolique, l'analyse de flot de donnés / le contrôle de flot d'information.

  La plupart de ces analyses sont aussi utilisées lors de la compilation pour déterminer certaines optimisations.

6. La définition classique met plus l'accent sur l'aspect "un 'mauvais' évènement n'arrive jamais"[3] alors que la définition *dense* se base sur le fait qu'une telle propriété ne peut être invalidée que par une trace infinie[1, section 2 et §2.4 précisément].

Néanmoins, ces méthodes sont sûres mais pas complètes. En effet, tous les programmes violant la propriété précisée seront détectés, mais certains programmes la respectant peuvent ne pas être reconnus en tant que tels.

- Considérer un modèle du programme et vérifier cette propriété sur ce modèle, qui peut être fini à l'inverse de l'exécution du programme (model-checking).
- Synthétiser un programme de façon déductive à partir des propriétés voulues[55]. Cette méthode récente a cependant des restrictions. Par exemple, elle ne permet pas d'exprimer des programmes ne terminant pas.
- Tenter d'apporter une preuve de façon automatique si la propriété peut être exprimée à l'aide d'une logique suffisamment simple pour être décidée (efficacement), à l'aide d'un (ré-)solveur.
- Apporter une preuve manuelle vérifiée, préférablement, de façon mécanique avec un assistant de preuve.

Toutes ces méthodes, exceptée la dernière, se font de manière automatique après avoir spécifié les propriétés/le modèle du programme.

### 3.4.2 Assistants de preuves

Dans le cas de programmes complexes comme un compilateur, la dernière démarche est abordée car il est préférable de donner d'abord une preuve de bout en bout et de générer un programme à partir de cette preuve.

En effet, ces assistants de preuves tirent parti d'un postulat appelé la correspondance (ou équivalence/isomorphisme) de *Curry-Howard*, faisant correspondre les programmes à la logique formelle (les preuves).

Ainsi, les assistants de preuves tels que Coq, Isabelle[7], F* ou Lean, permettent de former des preuves selon des logiques qui correspondent à des modèles de calculs. Et ainsi, rendent possible de manière sûre la génération d'un programme à partir d'une spécification et d'une preuve.

#### 3.4.2.1 Coq

La logique[8] employée par Coq est connue sous le nom de Calcul des Constructions ou *Calculus of Constructions*, d'où le nom donné à l'assistant de preuve[9]. C'est une logique constructive, c'est-à-dire qu'elle n'utilise pas d'axiomes pour lesquels elle ne peut pas de donner de preuve qui peut être calculée. Ainsi, certains axiomes de la logique classique (et d'autres logiques très répandues en mathématiques telle que la théorie des ensembles ZFC) comme la loi du milieu exclu fort ($\forall proposition P, P \oplus \ /P$) entrent dans cette catégorie et doivent donc être admis et ne peuvent plus être utilisés en dehors de la preuve d'un théorème. Utiliser certains axiomes ensemble, par exemple tout axiome de la logique classique avec l'axiome du choix unique, brise même la logique de Coq.

Coq n'est pas qu'un assistant de preuves. En fait lorsque l'on parle de Coq, on inclut en réalité trois langages :

- Gallina, le langage de programmation fonctionnel pur total[10] de Coq, servant à définir types, fonctions et théorèmes,
- Vernacular, le langage de manipulation de tactiques pour résoudre les preuves (à l'inversion de Gallina, il n'est pas total et une preuve mal écrite peut boucler à l'infini) et
- Ltac, le langage servant à définir les tactiques de résolutions (pouvant aussi être définies en Vernacular, mais de façon beaucoup moins expressive).

#### 3.4.2.2 SSReflect[32]

Le projet *When good components go bad* fait aussi grandement usage de la librairie *mathcomp-SSReflect*[11] qui donne l'opportunité d'utiliser de nombreuses réflexions entre des notions d'équivalences de propriétés et d'égalités entre booléens (notion calculable), un point sur lequel Coq seul fait clairement la distinction. C'est

7. Uniquement pour certaines classes de spécifications en logique d'ordre supérieur
8. Aussi considérée comme théorie des types
9. En réalité, la logique utilisée est le Calcul des Constructions (Co-)Inductives, en anglais *Caluculus of (Co-)Inductive Constructions*, *CiC*. De plus, le créateur du Calcul des Constructions, qui a également fait parti des premiers développeurs de Coq, s'appelle *M. Thierry Coquand*.
10. C'est-à-dire sans effet de bord et dont toutes les fonctions récursives (*Fixpoint*) doivent terminer pour être définies.
11. Et de quelques autres *math-comp* pour manipuler en Coq certains éléments de mathématiques comme les groupes finis et de nombreuses notions d'algèbre.

une librairie très utilisée pour pouvoir reproduire le raisonnement mathématique sur papier et elle a servi pour formaliser des théorèmes dont la preuve papier est notablement longue (plusieurs centaines de pages)[30, 31, 33]. Néanmoins elle rend l'extraction des programmes moins efficace car de nombreuses définitions sont gardées opaques[12] (cette librairie a été réalisée dans une optique de preuve plutôt que de construction de programmes).

### 3.4.3 *Randomized property-based testing* avec QuickChick

Certaines méthodes moins formelles (complètes mais non sûres) permettent de découvrir certaines violations des propriétés comme l'écriture de tests. Pour rendre cette tâche largement plus aisée, *QuickCheck*[19], un générateur de cas de tests à partir d'une propriété, a été créé pour le langage Haskell. Son succès a été tel que de nombreux ports ont été fait dans une foultitude de langages, dont Coq avec *QuickChick* qui de plus se repose sur une théorie formelle[54] et permet de générer des cas plus pertinents[44].

Désormais, nous avons les outils en main pour vérifier les propriétés d'un programme. Il nous reste enfin à comprendre la compilation vérifiée, une base sur laquelle s'appuie la compilation sécurisée.

## 3.5 Compilation vérifiée

Comme vous le savez, un compilateur est un programme permettant de traduire un langage (source) en un autre (cible). De façon plutôt raisonnable, on attend d'un compilateur qu'il traduise de façon correcte les programmes ; principalement, qu'il ne donne pas un programme incorrect à partir d'un programme correct. Pour paraphraser le cours de *M. Fréderic Dabrowski* et *M. Fréderic Loulergue* citant l'un des ouvrages de référence en compilation :

> *It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct ! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free ! Thus the most important objective in writing a compiler is that it is correct.*

Mais comment savoir si un compilateur est correct ? Comme évoqué brièvement auparavant (3.3.1) et en s'appuyant sur les notions vues précédemment, il faut exhiber une correspondance entre les sémantiques des langages source, cible(s) et intermédiaires si nécessaires. Elle est donnée généralement informellement par la spécification du langage puis doit être formalisée. Enfin, par ce biais, elle doit fixer les points nécessaires sur lesquels la spécification a été évasive (ordre d'évaluation ou autres comportements définis par l'implémentation/non spécifiés).

Ensuite, le reste de la création d'un compilateur sécurisé va consister en la génération de code cible/intermédiaire avec les optimisations (découpées en transformations explicites entre deux langages ou dans le même) et la preuve que les transformations conservent correctement la sémantique du programme.

Mais en quoi consiste à donner cette preuve de correction ? On veut que le compilateur préserve les traces/comportements entre les différents langages, ce qui revient à vérifier d'autres preuves plus simples. Il est possible de donner la preuve de correction du compilateur :

### 3.5.1 Simulation "en arrière" ou *backward simulation*

On peut montrer que pour tout programme source W, si W compilé produit une trace t, (écrit $W\downarrow \rightsquigarrow t$), alors W, au niveau source, doit produire cette trace t également. Ce type de preuve est appelée correction de compilation en arrière (*Backward Correct Compilation* ou *BCC*). Cette *BCC* peut alors être montrée à l'aide d'une simulation "en arrière",

S'il est possible de trouver cette simulation, alors la preuve est "finie" (à haut niveau). Néanmoins ce genre de simulation est dure à prouver. La plupart du temps, les expressions du langage source sont souvent compilées en plusieurs instructions dans le langage cible. Reconstruire le programme source original à partir de celui compilé nécessite donc d'inspecter l'exécution de plusieurs instructions du langage cible pour trouver une instruction du langage source, et les différentes optimisations peuvent compliquer bien plus la tâche.

C'est pourquoi une autre technique de preuve permet de trouver cette simulation.

12. Les différentes définitions et preuves de coq peuvent être compilées de manière transparente ou opaque, ce qui a un impact sur la génération du code et la vérification[46] de la preuve.

### 3.5.2 Simulation “en avant” ou *forward simulation*

À l'inverse, une simulation “en avant” est bien plus facile à trouver. Avec quelques autres hypothèses sur le bon sens des langages, cette simulation “en avant” implique l'existence d'une simulation “en arrière”.

Par bon sens des langages, on entend une hypothèse sur leur déterminisme : si le langage cible est déterministe, cela suffit. Aucun non-déterminisme, même provenant du monde extérieur n'est possible alors. On voit évidemment que c'est une notion assez forte et cette hypothèse peut être relaxée en deux autres beaucoup moins contraignantes : le langage source doit être réceptif[13] et le langage cible déterminé[14]. On ne donnera pas la preuve de cette affirmation, non-triviale, mais celle-ci est présentée un peu plus en détail par *M. Xavier Leroy* pendant sa présentation à *DeepSpec Summer School*[47, vidéo à 1 :25 :20][50, slide 215 (233 en réalité)] et est disponible dans l'article qu'il a proposé [60] pour une preuve papier et dans le développement de *CompCert* pour une preuve Coq[49, pretty printed][63, github].

13. *Receptive* ou *input total*, c'est-à-dire que si il accepte en entrée une certaine valeur (lors d'une lecture de l'environnement), alors il doit accepter toutes les valeurs possibles.

14. *Determinate*, c'est-à-dire que le langage n'a pas de non-déterminisme interne, et que tout le non-déterminisme provient forcément de l'extérieur, de l'environnement.

# Contexte du stage

## 4.1 Inria

*Inria*, Institut National de Recherche en Informatique et automatique, anciennement connu sous les noms de l'INRIA et l'IRIA (même acronyme sans "National") est un institut de recherche public français créé sous l'impulsion du président Charles de Gaulle et de Michel Debré dans le cadre du plan Calcul, en même temps d'autres companies privées, pour réagir à la dégringolade et rachat de *Bull*.

De cet institut proviennent plusieurs sujets de recherche, et nombre d'entre eux ont éclos pour devenir accessible au public sous forme de logiciel libre comme le langage *OCaml* (la variante de *ML* la plus utilisée), l'assistant de preuve *Coq* écrit en *OCaml*, l'alternative libre à *Matlab*, *Scilab* ; ou de logiciels dont le code est en accès libre comme *CompCert*[1].

C'est avec le *Centre National de Recherche* ou *CNRS* et le *Commissariat à l'énergie atomique et aux énergies alternatives* ou *CEA* l'un des instituts publics majeurs de recherche en informatique en France. *Inria* a aussi des partenariats avec de nombreuses équipes et pôles étrangers dont le plus connu est celui avec *Microsoft Research* (*MSR* pour faire court), ayant abouti à la création d'un laboratoire à Saclay et à de nombreux partenariats entre les équipes de *MSR* et d'*Inria* (dont *Prosecco*).

De nombreux sites de cet institut se trouvent en France comme à Rocquencourt historiquement (abritant uniquement le siège de l'institut désormais) mais par la suite à Bordeaux, Grenoble, Lille, Nancy, Rennes, Saclay, Sophia Antipolis et bien sûr Paris, où l'équipe que j'ai intégré se situe.

## 4.2 Équipe Prosecco

J'ai rejoint l'équipe *Prosecco* non pas par attrait de la boisson mais pour les sujets abordés : *Prosecco* signifiant "PROgramming SECurely with CryptOgraph".

Cette équipe, fondée par *M. Karthikeyan Bhargavan* et dirigé par ce dernier mais aussi par *M. Cătălin Hriţcu*, *M. Bruno Blanchet* et *M. Harry Halpin*, a désormais plusieurs axes :

- Le développement d'outils de vérification cryptographique et d'assistants de preuves/langages de programmation. *M. Bruno Blanchet* dirige les travaux sur des démonstrateurs automatiques de théorèmes sur protocoles dans les modèles symboliques –considérant le protocole comme reposant sur des primitives cryptographiques parfaites– et calculatoires –raisonnant sur la probabilité de pouvoir casser les différentes primitives cryptographiques à l'aide d'un raisonnement à base de sémantiques de jeux[15]– [14] appelées *ProVerif* et *CryptoVerif* respectivement.

  *M. Cătălin Hriţcu* supervise une partie du développement de *F** (en collaboration avec *MSR*), principalement le compilateur du langage. Pour donner une explication concise, F* (prononcé "F star") est un langage de programmation fonctionnel avec types dépendants (notés *refinement types*) et effets (c'est-à-dire des annotation sur les valeurs de retour de fonction pour préciser les effets de bord possibles tels que la divergence, la manipulation du tas, la levée d'exception, etc.). F* se sert de ces fonctionnalités pour aider à la vérification de programme et se sert d'une vérification déductive réalisée par *Z3*[53], un résolveur *SMT*. Depuis très peu, la vérification peut aussi se faire à l'aide de tactiques *SMT* ou définies en F*[51]. Les programmes sont vérifiés statiquement en F* lors de la vérification des types (*type-checking*) et peuvent être extrait en F# ou OCaml. De plus et si le programme n'utilise qu'un sous-ensemble clairement défini de F*, appelé Low*[56], il est possible de générer du code C avec KreMLin[41] garantissant une sûreté mémoire si elle a été prouvée.

1. Et utilisation libre selon un usage personnel, éducationnel ou de recherche

- La conception et vérification de logiciels cryptographiques (primitives, protocoles et librairies) avec des méthodes formelles. Le projet miTLS, rattaché au Projet *Everest* de *MSR*, vise à fournir une pile TLS de référence intégralement prouvée sûre[13], des primitives (HACL*[65], écrit et prouvé en Low* et déployé dans Firefox [11])[2], au protocole (vérifié à l'aide de *ProVerif* et *CryptoVerif*[12, 40]) en passant par la création des records[24].
- Et plus récemment, la définition et création d'une chaîne de compilation sécurisée à travers le projet *SECOMP*.

## 4.3 Projet *SECOMP*

Le projet *SECOMP*[28] vise à apporter une chaîne de compilation sécurisée pour des langages comme C. Ce projet bénéficie d'une bourse de départ *European Research Council* ou *ERC*.

La sécurité visée par cette chaîne de compilation est de garder le respect de propriétés que l'on pourrait avoir au niveau source (obtenues par une méthode formelle) même lorsque le contexte qui est en lien avec le programme compilé permet des attaques de bas niveau. Dans cette optique, la chaîne de compilation devra fournir des garanties de cette préservation d'un point de vue statique, mais doit aussi assurer le respect de cette sécurité de façon dynamique.

Pour arriver à ce résultat de façon efficace, il est nécessaire de reposer sur des mécanismes de protections qui ne sont malheureusement pas disponibles dans les architectures matérielles classiques comme *x86* ou *ARM*. L'architecture envisagée permettant d'arriver à nos fins consiste principalement en un processeur *RISC-V* et un moniteur d'étiquettes implémenté physiquement mais avec des règles définissables logiciellement. Mais certaines autres architectures comme une machine à capacités comme *CHERI*[25] sont sûrement exploitables. Cette facette du projet provient d'un projet *DARPA* nommé *CRASH/SAFE*[58] désormais repris en quelque sorte par le projet *SSITH/HOPE*[9], sous l'ombrelle du *DARPA* également. L'aspect de définition des règles logicielles, *micro-policies* est défini exhaustivement dans la thèse de *M. Arthur Azevedo de Amorim*[6] contenant les pointeurs nécessaires pour retrouver les recherches impliquées. Les acteurs impliqués dans cette recherche sont aussi mentionnés sur le site de *SECOMP*.

Pour cela, différents axes ont été envisagés sur la définition d'un tel compilateur, à base de *full abstraction* avec compartimentation[38, 37] puis à l'aide d'un nouveau critère (prouvable de façon générique) pour la compilation sécurisé visant uniquement à la préservation des propriétés en présence d'un contexte antagonique[2] dont les multiples notions de préservations ont été formalisées[1][3].

Une telle chaîne de compilation présenterait un avantage certain car pourrait donner beaucoup plus de garanties contres des attaques à distances permettant l'injection de code [4].

Si ce projet permet d'exprimer un langage utilisable dans un contexte réaliste comme le langage de programmation C, son inclusion dans un projet comme *Everest* serait un progrès immense en sécurité[5].

2. En vérifiant également le code assembleur généré pour certaines primitives cryptographiques[17] (recherche n'impliquant pas Prosecco).

3. En plus de la formalisation de ces préservations, des techniques de preuves pour celles-ci sont fournies ainsi que la preuve d'un englobement (*collapse*) de certaines préservations par d'autres. Plus explicitement, la préservation des hyperpropriétés (relationnelles) de vivacité englobe la préservation de toutes les hyperpropriétés (relationnelles, respectivement). Voir plus loin pour ces définitions.

4. Néanmoins, des attaques par injection de faute par laser briseraient ces garanties. Mais ce type d'attaque requiert d'avoir un accès physique au système exécutant le programme.

5. Ceci est une conjecture strictement personnelle et aucun projet explicite d'inclure un projet de *SECOMP* dans *Everest* n'est donné explicitement.

# SECOMP : secure compilation, ou *Efficient Formally Secure Compilation to a Tagged Architecture*

Le projet *SECOMP* vise à construire une chaîne de compilation sécurisée mais efficace. Il se concentre actuellement sur deux aspects :

- Définir précisément ce que l'on voudrait obtenir par la compilation sécurisée, et quels critères permettent de l'atteindre ;
- Présenter une chaîne de compilation "preuve de concept" remplissant l'un des critères utiles mais relativement faciles à prouver.

## 5.1 Objectifs de la compilation sécurisée

Qu'entend-on par compilation sécurisée ? Jusque-là, nous avons parlé de compilation vérifiée, c'est-à-dire qui va bien préserver la sémantique du programme entre les différents niveaux. Cette démarche permet de s'assurer que le programme va bien conserver des propriétés que l'on aura prouvé au niveau source, mais selon certaines conditions, principalement :

- La totalité du programme doit être compilée par ce compilateur et
- Le programme doit respecter ces propriétés dans sa totalité (donc même dans ses dépendances, à cause du point précédent).

Dans les systèmes extrêmement critiques où une défaillance est une question de vie ou de mort, cette démarche peut être abordée. Dans ce genre de systèmes –transports, gestion de centrale nucléaire, armement–, il est *possible* (mais pas systématique) d'avoir un programme *relativement* concis et pouvant fonctionner plutôt en circuit fermé. Par exemple avec un programme lancé sans OS ni réseau, ou avec un sur un système d'exploitation embarqué vérifié (dont au moins le noyau est vérifié [26, 39]), etc.

### 5.1.1 Pourquoi la compilation vérifiée ne suffit-elle pas ?

Néanmoins, ces conditions ne tiennent pas pour la quasi-totalité des logiciels. L'écrasante majorité des logiciels va s'appuyer sur un système d'exploitation et des bibliothèques. Et même en admettant que l'on puisse faire confiance à toutes les briques sur lesquelles on construit un logiciel, il est quasiment impossible de prouver l'intégralité de celui-ci !

Prenons le cas d'un explorateur internet : pour avoir une preuve allant de bout en bout, il faut prouver les différentes propriétés sur de nombreux composants du logiciel comme le moteur de rendu, de lignes de codes, un effort quasi[1]-impossible à achever.

Effectivement, dans le cas de navigateurs open-source comme Chromium[27] ou Firefox[65], la librairie cryptographique –ou au moins une partie– est vérifiée[2]. Mais une grande partie du reste du navigateur internet peut être compromise et les garanties apportées par la preuve sur le navigateur internet sont alors perdues.

1. Un effort pour réaliser un noyau vérifié de navigateur internet a été accompli[35], mais s'appuyant tout de même sur le moteur de rendu WebKit
2. la partie vérifiée de BoringSSL comprend X25519 et P-256, celle de NSS X25519, Ed25519, Chacha20, Salsa20 et d'autres encore

### 5.1.2 Précédents avec *full abstraction*

Comment faire alors pour conserver certaines propriétés même dans un tel contexte ? Différentes approches ont été abordées pour répondre à cela. La plupart des recherches menées jusque là s'appuyaient sur la notion de *Full Abstraction* évoquée plus tôt (3.3.2.4). Celle-ci, permet d'apporter à deux programmes une équivalence basée sur leurs traces appelée équivalence observationelle. La *fully abstract compilation* permet de pousser ce raisonnement sur les différentes transformations du programme, étant équivalent de façon observable.

Néanmoins, la *fully abstract compilation* permet de conserver beaucoup de propriétés dans un langage *memory-safe*, mais n'est pas adaptée dans des langages non-sûrs comme C ou C++[36]. De plus, certaines propriétés, comme la confidentialité d'une clé dans un protocole cryptographique (qui est une forme de non-interférence) ne sont pas incluses dans cette notion d'équivalence observationnelle.

En outre, si jamais il est possible d'exprimer cette confidentialité au travers de l'équivalence observationnelle, la propriété qui en résulte serait beaucoup trop forte. La preuve de cette propriété ainsi que les mécanismes pour conserver cette propriété lors de l'exécution seraient très complexes et coûteux[1, introduction]. Ce problème additionnel se retrouve aussi si l'on choisit de cibler un critère de compilation trop fort pour l'usage voulu.

## 5.2 Préservation de propriétés et critères de compilation / *Exploring Robust Property Preservation for Secure Compilation[1, introduction]*

Pour cette raison, le projet *SECOMP* tente d'apporter des notions de chaînes de compilation sécurisées à l'aide de différentes notions de préservation **robuste** des propriétés vues en 3.3.2. Lorsque l'on parle de préservation de propriété d'états/de traces ou d'hyperpropriété, cela signifie que si les états/traces/ensembles de traces (respectivement) appartiennent à la propriété (dans le sens général) au niveau source, alors ceux-ci appartiennent à la propriété au niveau cible. Dans cette préservation des propriété, il n'est fait mention que de programmes entiers. L'alternative robuste de la préservation des propriétés met l'accent sur le fait que ces programmes peuvent être mis en lien avec n'importe quel contexte cible ou source.

Pour donner une instance de ce type de préservation, voyons la préservation robuste de propriétés de traces (*Robust Trace Property Presevation*, ou RTP), formellement donnée par :

$$\begin{aligned}\text{RTP}: \quad & \forall \pi \in 2^{Trace}.\ \forall \mathrm{P}.\ (\forall \mathrm{C_S}\ t.\ \mathrm{C_S}\,[\mathrm{P}] \rightsquigarrow t \Rightarrow t \in \pi) \\ & \Rightarrow (\forall \mathbf{C_T}\ t.\ \mathbf{C_T}\,[\mathrm{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)\end{aligned}$$

Intuitivement, cette propriété signifie que pour toute propriété de trace, si un programme la respecte avec n'importe quel contexte considéré au niveau source, alors le programme compilé avec n'importe quel contexte au niveau cible la respectera également.

Cette propriété, que l'on appellera précisément *critère de compilation sécurisé*, permet d'avoir un raisonnement analogue à celui de la *full abstraction*, c'est-à-dire de pouvoir raisonner au niveau du langage source.

Pour chacun des critères de programmes, il existe une alternative "sans la propriété", c'est-à-dire que pour les observations (états, traces, ensembles de traces...) au niveau cible, il est toujours possible de les expliquer au niveau source.

Ainsi, l'équivalent de la RTP avec absence de propriété est :

$$\text{RTP}': \quad \forall \mathrm{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \mathbf{C_T}\,[\mathrm{P}{\downarrow}] \rightsquigarrow t \Rightarrow \exists \mathrm{C_S}.\ \mathrm{C_S}\,[\mathrm{P}] \rightsquigarrow t$$

Signifiant que pour tout programme, si, en présence d'un contexte au niveau cible arbitraire, ce programme produit une certaine trace, il est toujours possible de trouver un contexte au niveau source mis en lien avec le programme qui peut produire cette même trace.

Comme nous l'avons dit plus tôt (3.3.2.2.1), une propriété de sûreté est souvent plus facile à prouver qu'une propriété de vivacité (en tout cas c'est clairement le cas pour la réfuter). Il est possible de prouver les classes de propriétés de sûreté en se servant d'une fonction de *back-translation*, appliquée à un programme et ses exécutions (trace, ensemble de trace, etc.), c'est-à-dire sans perte de généralité, un ensemble fini de préfixes finis d'exécutions produit par ce programme et un contexte cible. En s'appuyant sur une fonction de *back-translation* permettant de récupérer un programme source, ce programme produit bien évidemment la même exécution. Ce point permet de prouver bien plus facilement ces critères de compilations et sera un peu plus détaillé par la suite (partie 5.3).

Cette technique permet de prouver le critère de préservation robuste de propriété de sûreté, ou RSP′ (la version "sans propriété") :

$$\text{RSP}' : \quad \forall \mathrm{P}.\ \forall \mathbf{C_T}.\ \forall m.\ \mathbf{C_T}\left[\mathrm{P}{\downarrow}\right] \rightsquigarrow m \Rightarrow \exists \mathrm{C_S}.\ \mathrm{C_S}\left[\mathrm{P}\right] \rightsquigarrow m$$

Déclarant que pour tout programme (compilé) P qui, mis avec un contexte cible arbitraire $\mathbf{C_T}$ , produit au niveau cible un préfixe fini de trace $m$, il est toujours possible de trouver un contexte source $\mathrm{C_S}$ qui produit ce même $m$ avec P au niveau source. Cette caractérisation est celle qui sera adaptée dans le compilateur décrit plus tard.

Ces différentes notions de préservations robustes ont été définies sur l'ensemble des différentes classes de propriétés. Mais il n'est pas préférable de viser le critère le plus généraliste possible car, comme avec la *full abstraction*, la preuve à fournir peut être plus complexe et les mécanismes requis pour imposer cette préservation à l'exécution beaucoup trop lourds. C'est d'ailleurs ce que nous allons voir par la suite avec le compilateur "preuve de concept", l'un des autre projets de *SECOMP*.

## 5.3 Un compilateur sécurisé simple / *When Good Components Go Bad[2]*

### 5.3.1 Compartimentation

Dans ce compilateur, le critère visé est RSP, appelé RSC (pour *Robustly Safe Compilation*) dans l'article, adapté pour pouvoir ajouter l'aspect "compartimentation" des programmes. Pour pouvoir assurer le respect de cette compartimentation à l'exécution, il a été choisi deux *back-ends* :

- Un environnement d'exécution à base d'isolation de faute logicielle (ou *Software Fault Isolation*, en court *SFI*), souvent connu sous le nom de *sandbox*. Ce système a l'avantage d'être utilisable sur n'importe quel ordinateur mais est plutôt lent, le problème étant que les restrictions sont vérifiées par le logiciel. Ce type de système est très utilisé en virtualisation et a été tenté dans Chromium pour exécuter des programmes natifs avec *NaCl* (*Native Client*)[64].

- Une architecture matérielle (*DOVER*) basé sur un processeur RISC-V avec un moniteur d'étiquettes implémenté matériellement à l'aide d'un FPGA[3] (comprenant un cache physique dédié)[23, 62] permettant d'imposer le respect de règles définies au niveau logiciel appelées *Micro-policies*[8, 6, 4]. Ces dernières peuvent exprimer, entre autre, une intégrité de flot de contrôle, un contrôle du flot d'information, une sûreté mémoire du tas/de la pile ou encore une compartimentation. *DOVER* et les *Micro-policies* ont été conçues et vérifiées avec des méthodes formelles (voir références précédentes).

Ce compilateur vise un langage simple mais *Turing-complet*, avec procédures, espaces mémoire (*buffers*) allouées statiquement ou dynamiquement, accès explicite à la mémoire (mais sans sûreté mémoire) et avec une compartimentation basée sur des composants et interfaces. Cela signifie qu'un programme peut être divisé en plusieurs unités indépendantes ayant une interface définissant les procédures que le composant exporte et importe. Dans le premier cas, les procédures peuvent être appelées depuis les autres et dans le second cas ce sont celles sur lesquelles le composant déclare demander l'accès, dont les appels systèmes donnant l'accès à l'environnement (I/O).

Ces composants possèdent chacun leurs espaces mémoire (représentés sous forme de blocs) et seul un de ceux-ci a son code exécuté à la fois (pas de concurrence pour l'instant). Un composant ne pouvait atteindre directement la mémoire, et le seul moyen de communication entre les différents composants était cette interface. Ces composants et leurs interfaces sont tous deux définis statiquement dans l'état actuel du développement.

### 5.3.2 Modèle d'attaquant (*Threat model*)

Ce langage, que l'on tente de rapprocher petit à petit du langage C, exhibe des comportements non-definis (*undefined behavior*) et on considère ceux-ci comme le moyen selon lequel un attaquant peut compromettre un composant. De surcroît, le compromis d'un composant ne peut se faire que par l'interface du programme[4], donc si un comportement ne rencontre aucun comportement indéfini (de façon formellement prouvée), alors il ne peut pas être compromis.

Ce compromis est défini selon "deux dimensions" :

3. *Field Programmable Gate Array*, un circuit intégré qui peut être configuré après sa fabrication.
4. Ce qui signifie l'ensemble des interfaces des composants.

- Spatialement, comme vu juste au dessus, le compromis est restreint au composant ayant rencontré un *undefined behavior*, sauf, si il arrive à déclencher un *undefined behavior* en envoyant de mauvaises données à un autre.

- Mais aussi temporellement : bien que le standard du C déclare qu'un programme rencontrant un *undefined behavior* n'a pas de sens tout simplement, il a été décidé que le programme a un sens jusqu'à ce qu'il le rencontre. Considérer ceci est possible grâce au fait que le compilateur, prenant cette propriété de *CompCert*, ne change pas l'ordre des évènements observables lors de la génération de code, et donc ne peut faire intervenir un *undefined behavior* au niveau cible plus tôt que dans le code source.

De cette façon, on considère qu'une fois qu'un composant est compromis, son code est réécrit et l'on "revient en arrière" (au début du programme) car les composants sont définis statiquement. Ensuite, le composant compromis voit son code réécrit (tout en gardant son interface, le respect de celle-ci est vérifiée à l'exécution par le *backend*) en reproduisant le préfixe de la trace jusqu'au point où le composant a rencontré un *undefined behavior*. Ensuite on peut considérer que le programme n'a plus a respecter le suffixe de la trace et peut contacter les autres composants lorsqu'il aura le droit d'agir. Si jamais un tel composant a une de ses procédures appelées, il renverra une valeur spéciale "non définie" (*undefined value*) qui va être non déterministe. Ainsi tous les cas sont envisagés du point de vue des composants non compromis.

### 5.3.3 Architecture

Ce compilateur va traduire l'*AST* d'un programme du langage source vers un langage intermédiaire proche d'une machine *RISC* mais gardant toujours la notion de composants et d'interfaces. Le code et la mémoire des différents composants sont clairement séparés et représentés par un tableau associatif (*Map*) allant de l'identifiant du composant vers deux *Map* contenant les procédures et les blocs (respectivement), indexées selon leur ordre de déclaration (d'allocation pour les blocs).

Ensuite, le programme en langage intermédiaire peut être compilé vers le *backend* utilisant la *SFI* ou les *micro-policies*. À ce niveau, la mémoire est considérée comme un grand tableau et la notion de composant est définie par une section sur ce tableau. Les accès mémoire et appels de fonctions sont vérifiés à l'exécution en fonction de celle ayant fait l'appel et de méta-données fournies pendant la dernière étape de compilation.

Comme nous l'avons dit précédemment, ce compilateur utilise un modèle de traces à base d'évènements inspirés de *CompCert*. Des sémantiques opérationnelles structurées sont fournies pour tous les langages. Une preuve sur un critère proche de RSP est faite entre le langage cible et le langage intermédiaire. La compilation entre le langage intermédiaire et les langages des *back-ends*, bien que non prouvée robuste, a été soumise à un test à base de propriété par *QuickChick*[54]. De plus, comme mentionné auparavant, les *micro-policies* permettent d'exprimer le respect de la compartimentation, les contextes au niveau cible non exprimables par un programme au niveau source étant rejetés à l'exécution.

### 5.3.4 Preuve de haut niveau

#### 5.3.4.1 Adaptations nécessaires

##### 5.3.4.1.1 Langage non-sûr

Le critère de compilation RSP est un critère de compilation pour les langages source sûrs (sûreté de type/mémoire), il a fallu concilier cette notion avec celle d'un langage non-sûr. Il en a résulté l'utilisation (et la formalisation) de la compartimentation pour rendre plus facile à isoler le côté non-sûr du langage.

RSP (RSC ) est formulé légèrement différemment dans cet article, sous la forme :

$$\mathrm{RSP}' : \quad \forall \mathrm{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \mathbf{C_T}\left[\mathrm{P}\downarrow\right] \rightsquigarrow t \Rightarrow \forall m \leq t.\ \exists \mathrm{C_S}.\ \exists t'.\ \mathrm{C_S}\left[\mathrm{P}\right] \rightsquigarrow t' \wedge m \leq t'$$

Tout d'abord, la façon de considérer le compromis comme pouvant arriver à un certain moment de l'exécution, appelée compromis dynamique[5] est la première brique pour adapter le critère RSP pour un langage non-sûr. L'introduction d'un évènement spécial $Undef$ modélisant un *undefined behavior* survenant dans la trace permet de trouver qui, entre le programme et le contexte, a déclenché un tel évènement.

L'introduction de cet évènement permet une première relaxation (en souligné) pour arriver au critère :

5. Contrairement au standard du C et des tentatives précédentes de compilation sécurisée[36] ne considérant qu'un compromis statique.

$$\mathrm{RSC}^{\mathrm{DC}} : \quad \forall \mathrm{P}.\, \forall \mathbf{C_T}.\, \forall t.\, \mathbf{C_T}\,[\mathrm{P}\!\downarrow] \rightsquigarrow t \Rightarrow \forall m \leq t.\, \exists \mathrm{C_S}.\, \exists t'.\, \mathrm{C_S}\,[\mathrm{P}] \rightsquigarrow t' \wedge m \leq t' \quad \underline{\vee \quad t' \prec_P m}$$

où $t' \prec_P m$ signifie que $t'$ peut être un préfixe de $m$ terminant par $Undef$.

#### 5.3.4.1.2 Compartimentation

En ajoutant la notion de compartimentation (composants et interfaces), le langage prend en compte une notion de composants avec des privilèges spécifiés (interfaces) et "mutuellement méfiants" (*mutually distrustful*) car communicant uniquement au travers des interfaces. Cela permet aux composants de se protéger les un des autres et encourage l'utilisation de composants requérant le moins de privilèges possibles.

Pour cela, il faut ajouter le fait que chaque composant est compilé à part et définir la fonction de *back-translation* $\uparrow$ donnant un composant source à partir de sa trace tel que le composant recréé respecte l'interface originale. Cela nous permet d'aboutir à :

$$\begin{aligned}\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}} : \quad & \forall \mathrm{P}{:}I_P.\, \forall \mathbf{C_T}{:}I_C.\, \forall t.\, \mathbf{C_T}\,[\mathrm{P}\!\downarrow] \rightsquigarrow t \Rightarrow \forall m{\leq}t. \\ & \exists t'.\, (\{(m, I_i)\uparrow \mid I_i \in I_C\} \cup \mathrm{P}) \rightsquigarrow t' \wedge (m{\leq}t' \vee t' \prec_{I_P} m).\end{aligned}$$

Ou plus intuitivement, $\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}}$ suit $\mathrm{RSC}^{\mathrm{DC}}$ mais en ajoutant le fait qu'un programme ou contexte corresponde à une interface plutôt que ne pouvoir diviser le programme entier $\mathbf{C_T}\,[\mathrm{P}\!\downarrow]$ que selon un programme $\mathrm{P}\!\downarrow$ et un contexte $\mathbf{C_T}$.

Dans cette écriture, $\mathrm{P}{:}I$ signifie que le programme $P$ satisfait l'interface $I$. De plus, contrairement aux critères précédents, l'ensemble des contextes $\mathrm{C_S}$ respectant leur interface $I_C$ est donné par la fonction $\uparrow$ appliquée aux préfixes $m$ et respectant les différentes interfaces des contextes $I_C$, tout en gardant le fait qu'un préfixe $m$ peut se terminer prématurément à cause d'un $Undef$.

### 5.3.5 $\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}} \Rightarrow$ RSCC

La façon informelle dont on a décrit le fonctionnement de la chaîne de compilation jusqu'ici est appelée RSCC [6].

Par chance, le critère que nous avons vu précédemment, $\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}}$ implique RSCC [2, théorème 3.5, page 7] ! Également, $\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}}$ est plus simple à prouver (la preuve en elle même est détaillée après le théorème précédemment cité et visible schématiquement en figure 4 du même article) et nécessite des hypothèses raisonnables sur la chaîne de compilation.

Pour survoler cette preuve et en introduisant les hypothèse quand elles sont nécessaires, en partant d'un programme (source) $\mathrm{P} : I_P$ compilé produisant, avec un contexte cible $\mathbf{C_T} : I_C$, le préfixe fini de trace $m$, autrement dit :

$\mathbf{C_T}\,[\mathrm{P}\!\downarrow] \rightsquigarrow m$ [7] :

- À partir de l'existence d'une fonction de *back-translation*, (hypothèse requise directement par les additions apportées par $\mathrm{RSC}^{\mathrm{DC}}_{\mathrm{MD}}$ (cf partie 5.3.4.1.2), il est possible de créer un programme $\mathrm{P}'$ avec un contexte source $\mathrm{C_S}$ produisant ce même préfixe :

  $(m, I_C, I_P)\uparrow = \mathrm{C_S}\,[\mathrm{P}'] \rightsquigarrow m$

- En s'appuyant sur une notion de correction de compilation en avant (*Forward Compiler Correctness*), il est possible de prouver que le programme et le contexte obtenus précédemment produisent la même trace une fois compilés :

  $\mathrm{C_S}\,[\mathrm{P}'] \rightsquigarrow m \Rightarrow \mathrm{C_S}\!\downarrow\,[\mathrm{P}'\!\downarrow] \rightsquigarrow m$

- Nous avions vu plus haut que les différents langages ont des sémantiques définies, c'est le cas d'une manière classique, c'est-à-dire en considérant le programme entier (*Complete Semantics* ou *CS*), mais aussi en considérant les composants isolés (*Partial Semantics* ou *PS*), par le biais d'une fonction de décomposition des programmes.

6. sa formalisation est disponible dans l'article

7. Bien entendu, un programme va consister en plusieurs composants, donc cette démarche va se faire sur l'ensemble des composants, chacun tour à tour étant le programme $P$ et l'ensemble des autres étant le contexte.

En supposant l'existence d'une telle fonction, il est possible de considérer que tous les programmes $P_T : I_P$ mis en lien avec des contextes au niveau cible $C_T : I_C$ émettant un certain préfixe de trace fini $m$ dans les sémantiques complètes peuvent être décomposés. Cela donne un programme isolé $P_T$ produisant le préfixe $m$ dans les sémantiques partielles, donc selon les évènements qu'il peut produire complétés par les évènements exprimables par l'interface $I_C$.

Cette décomposition est appliquée sur le programme original ($\mathsf{P}\downarrow$) et le contexte source reconstruit puis compilé ($\mathsf{C_S}\downarrow$ du point précédent).

- D'une façon complémentaire, si il existe une fonction de composition, il est possible de recréer un programme complet à partir de programmes partiels (et donc de pouvoir récupérer une trace dans les sémantiques complètes). On compose donc les deux programmes partiels obtenus par décomposition et on obtient le programme complet produisant toujours le même préfixe $m$ :

  $\mathsf{C_S}\downarrow \left[ \mathsf{P'}\downarrow \right] \rightsquigarrow m$

- Après avoir obtenu ce programme, en s'appuyant sur la notion classique de correction de compilateur (*BCC*) agrémentée de la possibilité d'avoir $Undef$ dans les traces (5.3.4.1.1) et de la compilation séparée des composants (5.3.4.1.2), il est possible de recréer un programme produisant le même préfixe $m$ ou ayant rencontré un $Undef$ :

  $\mathsf{C_S}\left[\mathsf{P}\right] \rightsquigarrow t \wedge (m \leq t \vee t \prec m)$

- Finalement, par toutes ces constructions, il ne reste plus qu'à montrer que le contexte recréé par *back-translation* ne contient pas d'$Undef$, point pouvant être montré avec les additions apportées par $\mathsf{RSC}^{\mathsf{DC}}$(5.3.4.1.1).

Ces hypothèses utilisées bout à bout impliquent $\mathsf{RSC}^{\mathsf{DC}}_{\mathsf{MD}}$.

# Sujet : vers le passage de pointeurs dans la compilation sécurisée

## 6.1 Réalisations techniques

### 6.1.1 Prolongement

L'état du projet au début de mon stage était donc le suivant : la preuve détaillée précédemment était en grande majorité prouvée et le langage en était à la description donnée, c'est-à-dire sans possibilité d'accès mémoire entre les différents composants.

### 6.1.2 Objectifs

#### 6.1.2.1 Prise en main

Tout d'abord, après avoir passé un temps non négligeable à apprendre toutes les notions nécessaires (voir infra 6.1.3), j'ai pu commencer à compléter certaines des preuves et faire un peu de refactoring pour mieux me familiariser avec le projet. J'ai comblé des preuves admises dans la partie *Source/Definability* (possibilité de définir la fonction de *back-translation*, première des hypothèses) et *Source/Composition* et réparé des preuves brisées sur la consistance de la machine intermédiaire dans *Intermediate/Machine*.

Également, une modélisation plus formelle de l'environnement avec procédures d'entrée/sortie (appels systèmes) à été envisagée et commencée pour aider cette prise en main, mais son importance à été revue à la baisse face à l'ajout du passage de pointeurs.

#### 6.1.2.2 Pistes envisagées

Le premier objectif était de rajouter le passage de pointeurs dans le langage. Mais, au vu de la complexité que cela représentait au niveau de la preuve et de la conception, il a été choisi de faire des modifications incrémentales.

En effet, les différents problèmes engendrés par le passage de pointeurs sont nombreux. Si ce dernier est fait de manière naïve, il ne sera pas possible d'utiliser les pointeurs autrement qu'en les repassant sans jamais les déréférencer.

Dans le cas contraire, cela pourrait entraîner des *undefined behavior*. Si on déréférence le pointeur pour lecture ou écriture, on risque de le faire en dehors de la mémoire attribuée. Il ne faut pas oublier que les composants peuvent "mentir" si ils sont compromis : il n'est pas possible de créer un faux pointeur au niveau source, mais dans l'état actuel, il est possible d'en créer un au niveau cible. C'est l'une des raisons pour lesquelles les pointeurs ne sont pas passables pour l'instant. Et ainsi, il serait possible de compromettre les autres composants autrement que par leurs interfaces.

Pour pouvoir réaliser le passage de pointeurs, des pistes ont été envisagées sur l'utilisation de capacités (*capabilities*)[29], mais compte tenu du temps restreint du stage après l'apprentissage nécessaire, il a été décidé de laisser cet aspect pour plus tard et de modifier le langage de façon incrémentale.

#### 6.1.2.3 Ajout de lecture de mémoire entre composants

Il a d'abord été choisi de rendre simplement toute la mémoire locale statique des composants visibles aux autres en lecture, en précisant la taille de celle-ci dans l'interface.

Néanmoins, cela aurait apporté son lot de problèmes au niveau de la *definability*. Pour rappel, cette technique repose sur une *back-translation* prenant la trace du programme exécuté et recrée un programme source respectant sa trace. Pour cela, il est nécessaire que l'interface soit respectée durant la *back-translation*. Cependant, la façon dont est implémentée la *back-translation*[2, section 4.3, *Back-Translation function*] repose sur l'utilisation d'un compteur permettant de savoir à quel point de l'exécution le programme (précisément, le composant retraduit) en est et donc quel évènement émettre. Dans sa version initiale, les interface ne comportent pas la taille de la mémoire statique d'un composant, dont celle-ci pouvait différer entre le composant originel et celui retraduit. Une simple modification naïve briserait ce point.

Il a été nécessaire de diviser la mémoire statique de chaque composant en deux parties : une publique, que n'importe quel composant peut utiliser à sa guise, et une privée, réservée au composant et utile lors de la *back-translation*. Cette mémoire privée n'a pas sa taille indiquée dans l'interface et contient le compteur utilisé dans la fonction de *back-translation*. Elle pourrait par la suite contenir les *capabilities* des différents pointeurs reçus ou envoyés (le fait que ces *capabilities* ne soient pas visibles est en effet appréciable).

Cette fonctionnalité permet le déréférencement d'un pointeur appartenant à un composant $C$ depuis le code d'un autre composant $C'$, une première étape dans le passage de pointeurs (même si cela se fait uniquement avec la mémoire statique).

#### 6.1.2.4 Conception

##### 6.1.2.4.1 Changements sur le langage

J'ai modifié la représentation de la mémoire des composants pour pouvoir avoir cette séparation *buffer* public/privé. Au passage, j'ai refactorisé le code que j'ai dû changer, précisément j'ai abstrait les types "primitifs" (NMap[1], etc.) utilisés et créé des fonctions pour récupérer les différents éléments (buffers d'un composant, ensemble des buffers publics, etc.) car la plupart des définitions actuelles utilisaient explicitement la représentation interne des buffers pour les manipuler. Ainsi si l'on désire changer de représentation pour la mémoire (map d'une autre librairie, autre structure de donnée), il est plus facile de pouvoir le faire. Néanmoins il serait préférable de refactoriser les preuves pour pouvoir appeler des lemmes propres aux structures de données.

Également, j'ai jugé bon de rajouter de nouvelles définitions pour assurer quelques propriétés sur les espaces mémoires[2], comme le respect de l'espace mémoire public alloué par rapport à son interface.

J'ai donc inséré une construction du langage pour avoir cette notion de pointeur appartenant à un autre composant par l'ajout d'expressions et de modifications des sémantiques au niveau du langage Source.

Comme l'accès à la mémoire d'un autre composant est un évènement observable, il a fallu aussi rajouter un nouveau type d'évènement, contenant l'adresse accédée en mémoire[3] et une valeur, sous la forme d'un entier ou non, définie si la mémoire lue n'a pas été initialisée ou si le composant a été compromis. Rajouter notre type *value* comme pouvant faire partie dans le type *Event* (et *trace*) a requis du code SSReflect pour ne pas changer le comportement des preuves sur ce point[4].

Ensuite, il a été nécessaire d'adapter les preuves brisées par ces modifications, ce qui représente la plus grande partie du temps de développement. Nombre de ces preuves ont été écrites avec SSReflect et ont été admises lorsqu'elles furent jugées trop longues à prouver par Coq mais simples à la main.

##### 6.1.2.4.2 Modification de la *back-translation*

La plupart des critères n'a pas eu a besoin d'être adaptée de façon complexe (*Composition*, *Decomposition*, les sémantiques partielles, etc.) mais la définition de la fonction de *back-translation* a requis plus de réflexion.

En effet, il adapter cette fonction pour exprimer l'évènement ajouté. Néanmoins, pour pouvoir émettre un tel évènement, les différents composants doivent avoir leur mémoire public remplie avec les bonnes valeurs avant d'être lues !

Pour pouvoir remplir la mémoire publique des valeurs étant lues, il faut regarder dans les évènements à venir de la trace (qui est un préfixe fini donc aucun problème de terminaison) pour connaître les valeurs qui seront éventuellement lues.

De cette façon, lorsque l'on arrive sur un évènement donnant le tour à un composant $C$ (ce qui entraînera une partie de l'écriture de son code), la *back-translation* va regarder le futur de la trace et regarder si un évènement de

1. La plupart des structures de données proviennent de la librairie standard de Coq, de *CompCert* et de *math-comp*, étendu par les librairies *extructures* et *coq-utils* de *M.Arthur Azevedo de Amorim*.

2. Il pourrait être aussi intéressant d'ajouter une définition pour vérifier si un déréférencement d'une mémoire se fait statiquement dans une partie allouée et définie de l'espace mémoire d'un autre composant (possible uniquement avec des indices déterminés statiquement).

3. Pour l'instant juste un offset car l'espace mémoire public est toujours alloué au bloc 0. Mais par la suite un pointeur pourra être renseigné pour rendre cet évènement compatible avec un accès sur n'importe quel espace mémoire partagé.

4. Il a fallu implémenter une *Canonical Structure* (presque équivalent de *typeclass* utilisé par SSReflect) et des preuves.

lecture de la mémoire de $C$ se produit et dans quelle case mémoire. Si c'est le cas, on affecte à cette case mémoire la valeur de l'évènement ($Undef$ ou $Intz$).

La *back-translation* va lire pour l'ensemble des cases allouées dans son buffer public, mais dès qu'un évènement concerne une case (et que l'on écrit dans la case idoine), il n'est plus nécessaire de regarder la suite de la trace pour cette case car elle peut être potentiellement réécrite.

On itère ainsi sur l'intégralité de la trace et des cases mémoires, et on réitère pour le prochain composant $C'$ ayant sa valeur lue.

#### 6.1.2.4.3 Exemple avec une nouvelle *back-translation*

Récupérons le code d'un programme par *back-translation* avec l'entrée suivante :

Listing 6.1 : Trace et interfaces

```
interface :=
    C1  {| import := [(C2, P1];
           export := [];
           size   := 0    |};
    C2  {| import := [];
           export := [P1]
           size   := 1    |};
buffers :=
    C1 , (public := [0],
# on ne se soucie pas de la définition de private tant
# qu'il respecte la taille minimale (1 pour le compteur)
         );
    C2 , (public := [1]
# idem
         );
Traces
# Rappel : l'évènement ELoad contient
#   le composant appelant
#   l'offset
#   la valeur
#   le composant dont on lit le buffer

  # On considère que le composant C1 commence
  # (composant principal)
      # lecture de l'entier 1
ev1:    ELoad   C1    0     1   C2
      # invisible dans la trace, C1 a changé
      # la valeur de son public[0] pour 42
      # (précédemment 0)
  # Appel de procédure -> Tour de C2
  # (pas d'importance dans l'argument dans notre modification)
ev2:    ECall   C1    P1    _1  C2
      # lecture de l'entier 1337
ev3:    ELoad   C2    0    1337 C1
  # Retour -> Tour de C1
  # (par d'importance pour le retour)
ev4:    ERet    C2          _2  C1
      # lecture de la réponse à la vie, l'univers
      # et tout le reste
ev5:    ELoad   C1    0     42  C2

# Fin du programme
```

On doit récupérer les programmes des composants suivants (respectant la trace et l'interface) :

Listing 6.2 : Programme retraduit

```
# même interface et buffers
```

```
# excepté éventuellement pour celui public de C1
# ( peut être rempli à l'avance)
      C1 , (public := [1337],
            (* private... *) );

#Code de C1
C1 {
  P1() {
    if (local.private[0] == 0) {
      local.private[0]++;
# Si non rempli à l'avance dans la définition du programme
# (pourrait rajouter des règles non nécéssaires :)
      # local.public[0] := 1337;
# déclenche ev1
      C2.local.public[0];
# déclenche ev2
      C2.P1(1);
# appel récursif pour décider à nouveau de l'action à faire
      C1.P1(0);
    } else if (local.private[0] == 1) {
      local.private[0]++;
# déclenche ev5
      C2.local.public[0];
      C1.P1(0);
    } else {
      exit();
    }
  }
}

# Code de C2
C2 {
  P1() {
    if (local.private[0] == 0) {
      local.private[0]++;
# changement de la valeur qui sera lue
      local.public[0] := 42
# déclenche ev3
      C1.local.public[0]
# déclenche ev4
      return 2;

    } else {
      exit();
    }
  }
}
```

Les modifications pour la *back-translation* ne sont pas encore implémentées mais les changements détaillés précédemment ont été validés.

### 6.1.3 Problèmes rencontrés

Tout d'abord, il m'a fallu un temps d'apprentissage très loin d'être négligeable avant de démarrer le projet à proprement parler. En effet, j'ai du apprendre Coq et les différentes notions spécifiques présentées dans l'introduction au domaine (et plus encore).

J'ai également du me familiariser au projet Coq lui-même, un compilateur pour un langage simple mais consistant en 20 000 lignes de codes et preuves. Mes superviseurs ont émis l'idée de résoudre quelques preuves simples pour me permettre de mieux m'approprier le projet, ce qui a porté ses fruits. Cela m'a aidé à appréhender

les différentes briques de base du projet et les types de lemmes les plus souvent utilisés.

De plus, certaines des parties du projet sont écrites à l'aide des librairies *math-comp*, ce qui change drastiquement le style des preuves et malheureusement réduit leur lisibilité. Également, les preuves en apparence simples en logique sur papier sont souvent plus difficiles à exprimer en Coq.

# Conclusion

Pour finir, j'ai pu apporter ma modeste contribution en étendant le langage sur un point important. À l'heure actuelle il reste encore à compléter ces améliorations. La fin de mon stage et début de contrat porteront sur ces changements et l'ajout de modifications par palier pour arriver au passage de pointeurs avec *capacités*. De plus, de nombreuses pistes pour étendre le projet sont envisagées comme le portage de la preuve de RSCC entre le langage intermédiaire et les langages des *back-ends*, l'utilisation d'autres back-ends, ou l'implémentation d'un compilateur utilisant la compartimentation mais sans une démarche aussi formelle pour voir comment cette notion peut s'adapter à un réel langage rapidement. La liste des pistes pour achever ce projet est longue et cette notion de compilation correcte est très jeune encore (le projet Coq n'est vieux que d'un an et demi).

J'ai pris beaucoup de plaisir à m'investir dans ce sujet malgré le fait d'avoir eu à passer une très grande partie de mon stage à apprendre toutes les bases nécessaires. Travailler durant ce stage dans un projet de recherche au sein d'une équipe passionnée a été une expérience très gratifiante et agréable. Également côtoyer de nombreux chercheurs investis dans leurs recherches et partageant leur engouement et leurs découvertes est quelque chose de très enrichissant.

À l'issue de ce stage, j'ai appris énormément sur les méthodes formelles et les preuves assistées de logiciel ; principalement à l'aide de tout ce que j'ai du apprendre pour mener à bien mon travail mais aussi pendant les différents séminaires assistés et l'école d'été à laquelle j'ai participé. Cela m'a permis d'élargir ma connaissance dans les domaines des méthodes formelles et de la cryptographie sur des sujets que je n'aurais peut-être pas aborder de mon propre chef.

J'ai pu me resservir d'une partie de mes connaissances acquises durant ma formation mais j'aurai apprécié pouvoir aborder plus en profondeur certaines parties vues en cours telles que les sémantiques et propriétés de langage de programmation, l'utilisation et le principe de Coq. Mais les notions de compilation, de logique, de model-checking et d'analyse statique que j'ai appris durant mon cursus et les deux projets de recherche du Master 2 m'ont grandement aidé pour ce stage.

# Bibliographie


[1] C. Abate et al. “Exploring Robust Property Preservation for Secure Compilation”. In : *ArXiv e-prints* (juil. 2018). arXiv : 1807.04603 [cs.PL].

[2] Carmine Abate et al. “When Good Components Go Bad : Formally Secure Compilation Despite Dynamic Compromise”. In : *ArXiv e-prints* (sept. 2018). url : https://arxiv.org/abs/1802.00588.

[3] Bowen Alpern et Fred B. Schneider. “Defining Liveness”. In : *Inf. Process. Lett.* 21.4 (1985), p. 181-185. doi : 10.1016/0020-0190(85)90056-0. url : https://www.cs.cornell.edu/fbs/publications/defliveness.pdf.

[4] Arthur Azevedo de Amorim et al. “A verified information-flow architecture”. In : *Journal of Computer Security* 24.6 (2016), p. 689-734. doi : 10.3233/JCS-15784. url : http://dx.doi.org/10.3233/JCS-15784.

[5] Andrew W Appel. “Verified software toolchain”. In : *European Symposium on Programming*. Springer. 2011, p. 1-17.

[6] Arthur Azevedo de Amorim. “A methodology for micro-policies”. Thèse de doct. University of Pennsylvania, 2017. url : http://www.seas.upenn.edu/~aarthur/thesis.pdf.

[7] Arthur Azevedo de Amorim, Cătălin Hriţcu et Benjamin C. Pierce. “The Meaning of Memory Safety”. In : *7th International Conference on Principles of Security and Trust (POST)*. 2018, p. 79-105. doi : 10.1007/978-3-319-89722-6_4. url : https://arxiv.org/abs/1705.07354.

[8] Arthur Azevedo de Amorim et al. “Micro-Policies : Formally Verified, Tag-Based Security Monitors”. In : *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, 2015, p. 813-830. isbn : 978-1-4673-6949-7. doi : 10.1109/SP.2015.55. url : http://prosecco.gforge.inria.fr/personal/hritcu/publications/micro-policies.pdf.

[9] *Baking Hack-Resistance Directly into Hardware*. url : https://www.darpa.mil/news-events/2017-04-10 (visité le 21/09/2018).

[10] Lennart Beringer et al. “Verified Compilation for Shared-Memory C”. In : *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Sous la dir. de Zhong Shao. T. 8410. Lecture Notes in Computer Science. Springer, 2014, p. 107-127. isbn : 978-3-642-54832-1. doi : 10.1007/978-3-642-54833-8_7. url : https://www.cs.princeton.edu/~appel/papers/shmemc.pdf.

[11] Benjamin Beurdouche et al. *HACL* in Mozilla Firefox: Formal methods and high assurance applications for the web*. Real World Crypto Symposium. 2018. url : https://rwc.iacr.org/2018/Slides/Beurdouche.pdf.

[12] K. Bhargavan, B. Blanchet et N. Kobeissi. “Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate”. In : *2017 IEEE Symposium on Security and Privacy (SP)*. Mai 2017, p. 483-502. doi : 10.1109/SP.2017.26.

[13] Karthikeyan Bhargavan et al. “Everest : Towards a Verified, Drop-in Replacement of HTTPS”. In : *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Sous la dir. de Benjamin S. Lerner, Rastislav Bodík et Shriram Krishnamurthi. T. 71. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 1 :1-1 :12. isbn : 978-3-95977-032-3. doi : 10.4230/LIPIcs.SNAPL.2017.1. url : http://drops.dagstuhl.de/opus/volltexte/2017/7119.

[14] Bruno Blanchet. “Security Protocol Verification : Symbolic and Computational Models”. In : *Proceedings of the First International Conference on Principles of Security and Trust*. POST'12. Tallinn, Estonia : Springer-Verlag, 2012, p. 3-29. isbn : 978-3-642-28640-7. doi : 10.1007/978-3-642-28641-4_2. url : http://dx.doi.org/10.1007/978-3-642-28641-4_2.

[15] Andreas Blass. “A game semantics for linear logic”. In : *Annals of Pure and Applied Logic* 56.1 (1992), p. 183-220. issn : 0168-0072. doi : https://doi.org/10.1016/0168-0072(92)90073-9. url : http://www.sciencedirect.com/science/article/pii/0168007292900739.

[16] Sandrine Blazy. “Formal semantics”. Habilitation à diriger des recherches. Université d'Evry-Val d'Essonne, oct. 2008. url : https://tel.archives-ouvertes.fr/tel-00336576.

[17] Barry Bond et al. “Vale : Verifying high-performance cryptographic assembly code”. In : *Proceedings of the USENIX Security Symposium*. 2017.

[18] *C11 final draft*. Rapp. tech. url : http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf.

[19] Koen Claessen et John Hughes. “QuickCheck : A Lightweight Tool for Random Testing of Haskell Programs”. In : *SIGPLAN Not.* 46.4 (mai 2011), p. 53-64. issn : 0362-1340. doi : 10.1145/1988042.1988046. url : http://doi.acm.org/10.1145/1988042.1988046.

[20] Michael R Clarkson et Fred B Schneider. “Hyperproperties”. In : *Journal of Computer Security* 18.6 (2010), p. 1157-1210.

[21] Sylvain Conchon, Jérôme Feret et Xavier Rival. *Course material on Semantics and applications to verification*. Fév. 2017. url : https://www.di.ens.fr/~rival/semverif-2017/.

[22] Frédéric Dabrowski et Frédéric Loulergue. *Cours de compilation, master Informatique de l'université d'Orléans*. 2016.

[23] André DeHon et al. “DOVER: A Metadata-Extended RISC-V”. In : *RISC-V Workshop*. Accompanying talk at http://youtu.be/r5dIS1kDars. 2016. url : http://riscv.org/wp-content/uploads/2016/01/Wed1430-dover_riscv_jan2016_v3.pdf.

[24] A. Delignat-Lavaud et al. “Implementing and Proving the TLS 1.3 Record Layer”. In : *2017 IEEE Symposium on Security and Privacy (SP)*. Mai 2017, p. 463-482. doi : 10.1109/SP.2017.58.

[25] *Department of Computer Science and Technology : Capability Hardware Enhanced RISC Instructions (CHERI)*. url : https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/ (visité le 21/09/2018).

[26] Philip Derrin et al. “Running the Manual : An Approach to High-assurance Microkernel Development”. In : *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell '06. Portland, Oregon, USA : ACM, 2006, p. 60-71. isbn : 1-59593-489-8. doi : 10.1145/1159842.1159850. url : http://doi.acm.org/10.1145/1159842.1159850.

[27] Andres Erbsen et al. “Simple High-Level Code For Cryptographic Arithmetic–With Proofs, Without Compromises”. In : *Proceedings of the IEEE Symposium on Security and Privacy 2019 (S&P'19)*. IEEE. Mai 2019. url : http://adam.chlipala.net/papers/FiatCryptoSP19/FiatCryptoSP19.pdf.

[28] *ERC SECOMP - Efficient Formally Secure Compilation to a Tagged Architecture*. url : https://secure-compilation.github.io/ (visité le 21/09/2018).

[29] Edward F. Gehringer. “Capability-based Addressing”. In : *Encyclopedia of Computer Science*. Chichester, UK : John Wiley et Sons Ltd., 2003, p. 194-196. isbn : 0-470-86412-5. url : http://dl.acm.org/citation.cfm?id=1074100.1074197.

[30] Georges Gonthier. “Formal proof–the four-color theorem”. In : *Notices of the AMS* 55.11 (2008), p. 1382-1393.

[31] Georges Gonthier. “The Four Colour Theorem : Engineering of a Formal Proof”. In : *Computer Mathematics*. Sous la dir. de Deepak Kapur. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 333-333. isbn : 978-3-540-87827-8.

[32] Georges Gonthier, Assia Mahboubi et Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France, 2016. url : https://hal.inria.fr/inria-00258384.

[33] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In : *Interactive Theorem Proving*. Sous la dir. de Sandrine Blazy, Christine Paulin-Mohring et David Pichardie. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 163-179. isbn : 978-3-642-39634-2.

[34] ISO/IEC. *ISO/IEC 9899 :2011 - Programming languages – C*. 2011.

[35] Dongseok Jang, Zachary Tatlock et Sorin Lerner. “Establishing Browser Security Guarantees Through Formal Shim Verification”. In : *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. Bellevue, WA : USENIX Association, 2012, p. 8-8. url : http://dl.acm.org/citation.cfm?id=2362793.2362801.

[36] Y. Juglaret et al. “Beyond Good and Evil : Formalizing the Security Guarantees of Compartmentalizing Compilation”. In : *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. Juin 2016, p. 45-60. doi : 10.1109/CSF.2016.11.

[37] Yannis Juglaret et al. “Beyond Good and Evil : Formalizing the Security Guarantees of Compartmentalizing Compilation”. In : *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, p. 45-60. doi : 10.1109/CSF.2016.11. url : https://doi.org/10.1109/CSF.2016.11.

[38] Yannis Juglaret et al. “Towards a Fully Abstract Compiler Using Micro-Policies : Secure Compilation for Mutually Distrustful Components”. In : *CoRR* abs/1510.00697 (2015). url : http://arxiv.org/abs/1510.00697.

[39] Gerwin Klein et al. “seL4 : Formal Verification of an OS Kernel”. In : *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA : ACM, 2009, p. 207-220. isbn : 978-1-60558-752-3. doi : 10.1145/1629575.1629596. url : http://doi.acm.org/10.1145/1629575.1629596.

[40] Nadim Kobeissi, Karthikeyan Bhargavan et Bruno Blanchet. “Automated Verification for Secure Messaging Protocols and Their Implementations : A Symbolic and Computational Approach”. In : *2nd IEEE European Symposium on Security and Privacy*. Paris, France, avr. 2017, p. 435-450. doi : 10.1109/EuroSP.2017.38. url : https://hal.inria.fr/hal-01575923.

[41] *KreMLin is a tool for extracting low-level F* programs to readable C code : FStarLang/kremlin*. original-date : 2016-05-27T23 :29 :05Z. 20 sept. 2018. url : https://github.com/FStarLang/kremlin (visité le 21/09/2018).

[42] Ramana Kumar et al. “CakeML: a verified implementation of ML”. In : *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2014, p. 179-192. isbn : 978-1-4503-2544-8. doi : 10.1145/2535838.2535841. url : https://cakeml.org/popl14.pdf.

[43] Leslie Lamport et Fred B. Schneider. “Formal Foundation for Specification and Verification”. In : *Distributed Systems : Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*. Sous la dir. de Mack W. Alford et al. T. 190. Lecture Notes in Computer Science. Springer, 1984, p. 203-285. isbn : 3-540-15216-4. doi : 10.1007/3-540-15216-4_15. url : https://doi.org/10.1007/3-540-15216-4_15.

[44] Leonidas Lampropoulos, Zoe Paraskevopoulou et Benjamin C. Pierce. “Generating Good Generators for Inductive Relations”. In : *Proc. ACM Program. Lang.* 2.POPL (déc. 2017), 45 :1-45 :30. issn : 2475-1421. doi : 10.1145/3158133. url : http://doi.acm.org/10.1145/3158133.

[45] Chris Lattner. *What Every C Programmer Should Know About Undefined Behavior #1/3*. LLVM Project Blog. 2011. url : http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[46] Xavier Leroy.

[47] Xavier Leroy. *DeepSpec Summer School, Part 16, Leroy (July 18, 2017)*. url : https://www.youtube.com/watch?v=C8F-vJgtYgQ (visité le 20/09/2018).

[48] Xavier Leroy. “Formal verification of a realistic compiler”. In : *Commun. ACM* 52.7 (2009), p. 107-115. doi : 10.1145/1538788.1538814. url : http://doi.acm.org/10.1145/1538788.1538814.

[49] Xavier Leroy. *Pretty printed doc of SmallStep module of CompCert*. url : http://compcert.inria.fr/doc/html/compcert.common.Smallstep.html.

[50] Xavier Leroy. *The formal verification of compilers*. 2017. url : https://xavierleroy.org/courses/DSSS-2017/slides.pdf.

[51] Guido Martínez et al. *Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms*. arXiv :1803.06547. Juil. 2018. URL : https://arxiv.org/abs/1803.06547.

[52] Andrew McCreight, Tim Chevalier et Andrew P. Tolmach. "A certified framework for compiling and executing garbage-collected languages". In : *15th ACM SIGPLAN International Conference On Functional Programming*. Sous la dir. de Paul Hudak et Stephanie Weirich. ACM, 2010, p. 273-284. ISBN : 978-1-60558-794-3. DOI : 10.1145/1863543.1863584. URL : http://doi.acm.org/10.1145/1863543.1863584.

[53] Leonardo de Moura et Nikolaj Bjørner. "Z3 : An Efficient SMT Solver". In : *Tools and Algorithms for the Construction and Analysis of Systems*. Sous la dir. de C. R. Ramakrishnan et Jakob Rehof. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 337-340. ISBN : 978-3-540-78800-3.

[54] Zoe Paraskevopoulou et al. "Foundational Property-Based Testing". In : *6th International Conference on Interactive Theorem Proving (ITP)*. T. 9236. Lecture Notes in Computer Science. Springer, 2015, p. 325-343. ISBN : 978-3-319-22101-4. DOI : 10.1007/978-3-319-22102-1_22. URL : http://prosecco.gforge.inria.fr/personal/hritcu/publications/foundational-pbt.pdf.

[55] N. Polikarpova et I. Sergey. "Structuring the Synthesis of Heap-Manipulating Programs". In : *ArXiv e-prints* (juil. 2018). arXiv : 1807.07022 [cs.PL].

[56] Jonathan Protzenko et al. "Verified Low-Level Programming Embedded in F*". In : *PACMPL* 1.ICFP (2017), 17 :1-17 :29. DOI : 10.1145/3110261. URL : http://arxiv.org/abs/1703.00053.

[57] John Regehr. *A Guide to Undefined Behavior in C and C++, Part 3*. Embedded in Academia blog. 2010. URL : https://blog.regehr.org/archives/232.

[58] *SAFE project*. URL : http://www.crash-safe.org/.

[59] Ilya Sergey. *Concurrent Separation Logics family tree*. URL : http://ilyasergey.net/other/CSL-Family-Tree.pdf.

[60] Jaroslav Sevcík et al. "Relaxed-memory concurrency and verified compilation". In : *Symposium on Principles of Programming Languages (POPL)*. ACM, 2011, p. 43-54. URL : http://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/paper.pdf.

[61] Gordon Stewart et al. "Compositional CompCert". In : *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015, p. 275-287. ISBN : 978-1-4503-3300-9. DOI : 10.1145/2676726.2676985. URL : https://www.cs.princeton.edu/~appel/papers/compcomp.pdf.

[62] G. T. Sullivan et al. "The Dover inherently secure processor". In : *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. Avr. 2017, p. 1-5. DOI : 10.1109/THS.2017.7943502.

[63] CompCert Team. *Github page of SmallStep module of CompCert*. URL : https://github.com/AbsInt/CompCert/blob/master/common/Smallstep.v.

[64] Bennet Yee et al. "Native Client: a sandbox for portable, untrusted x86 native code". In : *Communications of the ACM* 53.1 (2010), p. 91-99. URL : http://research.google.com/pubs/archive/34913.pdf.

[65] Jean-Karim Zinzindohoué et al. "HACL* : A Verified Modern Cryptographic Library". In : *CCS*. 2017.