Contrôle sûr de chaînes d'obfuscation logicielle

Nicolas Szlifierski

DOCTORAT BRETAGNE LOIRE MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

l'École Nationale Superieure Mines-Telecom Atlantique
Bretagne Pays de la Loire - IMT Atlantique

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Nicolas SZLIFIERSKI

Contrôle sûr de chaînes d'obfuscation logicielle

Thèse présentée et soutenue à IMT Atlantique, Brest, le 17/12/2020
Unité de recherche : Lab-STICC
Thèse N° : 2020IMTA0223

Rapporteurs avant soutenance :

Catherine DUBOIS	Professeur, ENSIIE
Pierre-Etienne MOREAU	Professeur, Mines Nancy – Université de Lorraine

Composition du Jury :

Président :	Thomas JENSEN	Directeur de recherche, INRIA
Examinateurs :	Fabien DAGNAT	Maître de conférences, IMT Atlantique
	Catherine DUBOIS	Professeur, ENSIIE
	Laure GONNORD	Maîtresse de conférences, Université Claude Bernard Lyon1
	Serge GUELTON	Ingénieur, Redhat
	Pierre-Etienne MOREAU	Professeur, Mines Nancy – Université de Lorraine
Dir. de thèse :	Fabien DAGNAT	Maître de conférences, IMT Atlantique

Invité :

Sébastien JOSSE	Chercheur, DGA Maîtrise de l'information

Cette thèse a été financée par la Direction Générale de l'Armement dans le cadre du Pôle d'Excellence Cyber.

Remerciements

Je tiens à remercier ici les différentes personnes sans qui mon doctorat et la rédaction de cette thèse n'auraient pas pu aboutir.

Je souhaite d'abord remercier particulièrement les rapporteurs et les membres du jury de ma thèse pour leur lecture de ce manuscrit et les remarques qu'ils ont apportées. Je souhaite également remercier Fabien Dagnat, mon directeur de thèse, et Serge Guelton pour leurs conseils et leur soutien tout au long de mon doctorat. Je remercie également Jérémy Buisson et Alain Darte, membres de mon comité de suivi, pour les différentes remarques et discussions au sujet de la thèse. Je tiens aussi à remercier les Écoles de Saint-Cyr Coëtquidan et mes collègues là-bas qui m'ont permis de terminer la rédaction de cette thèse.

Je remercie également mes collègues à IMT Atlantique. Je pense notamment aux doctorants avec qui j'ai eu d'innombrables discussions autour de cafés, et particulièrement, Etienne Louboutin avec qui j'ai partagé un bureau pendant toutes ces années. Je remercie également Benjamin Somers, qui a participé à l'implémentation du prototype présenté dans cette thèse lors d'un stage.

Ensuite, je souhaite remercier mes amis et les *trolls* d'IMT Atlantique pour la motivation qu'ils m'ont apportée. Enfin, je remercie ma famille, notamment mes parents et ma grand-mère pour leur soutien tout au long de ce doctorat.

Table des Matières

Chapitre 1

Introduction

Un programme est généralement très différent entre le code source écrit par le développeur et le code exécuté par le processeur de l'utilisateur final. Le code exécuté par le processeur ne sera pas le même que le code distribué à l'utilisateur final qui lui même est une transformation du code écrit par le développeur, qui est potentiellement distinct du code donné au compilateur (s'il y en a un). En effet, le code d'un programme est transformé plusieurs fois entre le moment où il est écrit et son exécution.

La transformation la plus évidente est celle réalisée par un compilateur, la transformation du code source dans un langage de programmation vers des instructions exécutables par le processeur de l'utilisateur du programme. Mais, même pour les langages qui n'utilisent pas de compilateur, le code distribué à l'utilisateur est souvent différent de celui écrit par le développeur. En effet, certains langages interprétés effectuent une étape de transformation sur le code avant d'être exécutés par la machine virtuelle. Par exemple, l'interpréteur standard CPython [47] convertit le code source Python en *bytecode* Python lors de la première exécution. Le développeur d'un programme Python peut distribuer soit le code source soit le *bytecode* à l'utilisateur. De même, les programmes Java sont compilés en *bytecode* Java puis sont ensuite exécutés par la machine virtuelle Java [88]. Même les programmes qui sont distribués dans le langage dans lequel ils sont écrits sont souvent transformés avant d'être distribués. Par exemple, les scripts JavaScript contenus dans les sites web sont le plus souvent minifiés [177] pour réduire la taille des fichiers transférés sur le réseau. De plus, dans la grande majorité des cas, le code du programme écrit est différent du code exécuté, car les processeurs ne sont pas capables d'exécuter la plupart des langages de programmation [1]. Enfin, le code exécuté peut même changer pendant l'exécution du programme : certaines machines virtuelles utilisent la compilation à la volée [125] pour rendre plus performante l'exécution de portions de code qui sont exécutées très souvent [139].

On peut identifier plusieurs types de transformations qui sont réalisées sur le code d'un

1. Avec quelques rares exceptions peu utilisées [182][98]

programme entre son développement et son exécution. Les premières sont les transformations qui traduisent le code d'un langage source vers un langage destination. Par exemple, la traduction du code source d'un programme C vers de l'assembleur. Cette traduction se fait même généralement en plusieurs étapes. GCC utilise plusieurs langages intermédiaires dans le processus de compilation (GENERIC, GIMPLE, RTL, ...) [150][163]. CLang utilise la représentation intermédiaire LLVM durant la compilation de programmes [133]. Enfin, le compilateur vérifié CompCert utilise 11 langages intermédiaires pour la compilation de programmes C [140]. Ces transformations peuvent potentiellement également détecter des erreurs (de syntaxe, de type ...) et générer des avertissements pour le développeur.

Un deuxième type de transformations sont les transformations qui ont pour objectifs d'améliorer les performances d'un programme. Les performances peuvent être la vitesse d'exécution [147], la taille sur le disque du programme [2], l'utilisation de la mémoire [201], le niveau de sécurité contre diverses attaques [71][156] ... Ces transformations consistent à modifier le code d'un programme pour améliorer les performances selon un ou plusieurs axes. En général, ces transformations conservent la sémantique du programme, c'est-à-dire que le comportement du programme ne change pas, mais certaines transformations peuvent modifier le comportement observable. Par exemple, certaines transformations d'optimisation utilisent les comportements non définis [2] pour améliorer les performances du programme [198]. Les transformations de sécurité peuvent également volontairement changer le comportement du programme, lorsque le programme détecte qu'il subit une attaque. Par exemple, si le programme détecte qu'il est dans un état (valeurs des variables, fonctions exécutées...) qui n'est pas possible dans une utilisation normale du programme [66]. Dans ce cas, le programme peut changer son comportement pour éviter l'attaque ou encore alerter le développeur de l'attaque [13], ces techniques sont généralement appelées *tamperproofing* [161].

Les programmes contiennent souvent de la propriété intellectuelle, en particulier des algorithmes, ou des mécanismes pour limiter l'utilisation du programme (système de licence ou de DRM (*Digital Rights Management*)). Étant donné que le développeur fournit un programme exécutable à l'utilisateur, un utilisateur malicieux pourra récupérer le code du programme. Il pourra utiliser ce code pour essayer d'accéder aux secrets ou de contourner les mécanismes de protection. Comme nous l'avons indiqué, le code auquel l'utilisateur a accès (celui de l'exécutable fourni aux utilisateurs) est différent du code source écrit par le développeur, mais, sans protection supplémentaire, un attaquant expérimenté ou

2. C'est-à-dire des constructions dont le comportement n'est pas défini par le langage

utilisant des outils avancés peut extraire autant d'informations (ou presque) de ce code que du code source original. Pour contrer ces attaques, l'objectif est de rendre le code auquel l'attaquant a accès difficilement compréhensible par un humain (éventuellement aidé d'outils automatiques). Il n'est pas vraiment pratique d'écrire un code directement incompréhensible, cela rendrait les tâches de développement, de maintenances et de mise à jour extrêmement difficile. Mais il est possible d'utiliser des transformations de code qui augmente la difficulté de compréhension du code, c'est le principe de l'obfuscation de code [72].

La transformation complète du code du programme entre la source et le programme exécutable demande donc d'appliquer plusieurs de ces transformations successivement. Cela peut amener plusieurs problématiques, par exemple il peut être intéressant d'assurer la correction des transformations de traduction et d'optimisation, c'est-à-dire qu'elles conservent bien la sémantique du programme. Ces transformations peuvent avoir des conditions d'applications assez précises qui doivent être vérifiées pour s'exécuter correctement, ou il peut y avoir des incompatibilités entre les transformations. L'enchaînement des transformations de compilation et d'optimisation a déjà beaucoup été étudié depuis longtemps et les outils (les compilateurs en particulier) sont bien adaptés à l'exécution de chaînes d'optimisation efficacement et en respectant les contraintes de ces transformations. Mais les transformations de sécurité (y compris les transformations d'obfuscation) ont des contraintes supplémentaires [3] par rapport aux transformations d'optimisation, et les compilateurs actuels retirent ou réduisent souvent les protections ajoutées (manuellement ou automatiquement) aux programmes [77]. Cependant, l'étude des compilateurs adaptés à la sécurité est beaucoup plus récente et beaucoup moins complète [190][59].

Dans cette thèse, nos objectifs vont être de proposer des solutions aux différentes problématiques de la compilation et de l'obfuscation de programme. En particulier, nous allons proposer des solutions pour faciliter la conception de chaînes de compilation prenant en compte l'obfuscation et, plus généralement, la sécurité des programmes. Pour cela, le but de la thèse sera de formaliser certains mécanismes des transformations de compilation et d'obfuscation, et leurs compositions, pour permettre un contrôle plus fin de l'application des transformations, de vérifier que les différentes contraintes des transformations sont respectées, ainsi que de fournir des informations sur le niveau et la couverture de la protection apportée aux développeurs.

Pour réaliser ces objectifs, cette thèse va présenter plusieurs contributions. Nous allons

3. Celles-ci sont détaillées dans les chapitres suivants

définir les concepts fondamentaux des chaînes d'obfuscation, tels que les transformations d'obfuscation et la manière dont elles sont composées. Nous allons également présenter des règles de conceptions pour les transformations et pour leurs compositions. Cela sera utilisé pour créer SCOL, un langage dédié à la conception et l'exécution de chaînes de compilation contenant des transformations d'obfuscation, qui permet de définir finement l'application des différentes transformations. De plus, nous allons définir un système de types pour ce langage permettant notamment de vérifier que les contraintes d'application des transformations sont respectées. L'utilisation d'un langage dédié permettra notamment aux utilisateurs de notre langage de créer des scripts grâce à des abstractions de haut niveau, sans connaissances avancées sur le fonctionnement interne du compilateur. Enfin, nous proposerons un prototype pour l'implémentation de ce langage et de son système de types et son intégration à l'infrastructure de compilation de LLVM.

Dans le chapitre 2, nous présenterons différentes techniques de protection de logiciels, des exemples de transformations, leurs spécificités et leurs contraintes. Puis, dans le chapitre 3, nous présenterons le principe de fonctionnement des chaînes de compilation, les particularités des chaînes d'obfuscation et les limites des compilateurs traditionnels pour exécuter ces chaînes. Dans ce chapitre, nous présenterons également plusieurs approches possibles pour résoudre ces problématiques. Dans le chapitre 4, nous présentons les sémantiques formelles de langages fonctionnels et de systèmes de type qui serviront de base pour la création de SCOL, notre langage dédié à la conception de chaîne de compilation et d'obfuscation. Ensuite, dans le chapitre 5, nous allons présenter globalement notre approche pour la conception et l'exécution des chaînes d'obfuscation, et comment les différentes contributions de cette thèse sont utilisées. Puis, dans le chapitre 6, nous présenterons SCOL, notre langage fonctionnel dédié aux chaînes d'obfuscation et ses différentes sémantiques. Dans le chapitre suivant, le chapitre 7, nous présenterons le système de types de SCOL qui participe notamment à la vérification de la correction des chaînes d'obfuscation. Enfin, dans le chapitre 8, nous présentons le fonctionnement de l'exécution du script d'obfuscation SCOL dans le gestionnaire de passes de LLVM. Finalement, dans la conclusion (chapitre 9), nous présenterons les limites et les perspectives des contributions que nous avons présentées précédemment.

Chapitre 2

Obfuscation et autres techniques de protection de logiciels

2.1 Introduction

Un programme peut contenir des secrets que le développeur ne veut pas dévoiler. Par exemple : le programme peut disposer d'un algorithme performant qui ne doit pas tomber dans les mains de ses concurrents. Le programme peut également contenir des clés cryptographiques, ou un protocole de communication particulier, qui permettent de communiquer avec un serveur. La connaissance de ces clés cryptographiques ou de ce protocole pourrait permettre à une tierce personne de créer un client alternatif entrainant une perte de revenus pour le développeur original. Enfin, le développeur peut vouloir également s'assurer que le comportement du programme ne peut pas être modifié, par exemple pour tricher dans des jeux vidéos, ou pour contourner des vérifications de licence ou de droits d'accès à du contenu.

Lorsque le programme est distribué, le développeur perd, au moins en partie, le contrôle de l'environnement dans lequel est exécuté le programme. Donc les utilisateurs peuvent étudier le programme librement et essayer d'en extraire les secrets. Il ne s'agit pas d'un problème de cryptographie, on ne peut pas simplement chiffrer le secret dans le programme et empêcher son extraction. En effet, le programme doit pouvoir s'exécuter, et lorsque le programme s'exécute le secret devra généralement apparaitre en clair à un moment.

Ce type d'attaques, effectuées sur le programme par l'utilisateur final du programme sont appelées *Man-At-The-End* (MATE), à l'instar des attaques réalisées par interceptions des communications entre un client et un serveur (*Man-In-The-Middle*).

Il existe plusieurs techniques de protection contre ce type d'attaque, notamment le *tamperproofing* et l'obfuscation. Nous verrons dans ce chapitre quelles sont les possibilités théoriques de l'obfuscation, et quelles sont les techniques utilisées en pratique et leurs limitations.

2.2 Attaques MATE

2.2.1 Introduction

Contrairement aux attaques *Man-In-The-Middle* [84], où l'on s'intéresse un attaquant qui intercepte les communications entre deux parties de confiance, dans les attaques *Man-At-The-End* (MATE) [4][70], on considère l'utilisateur final d'un programme comme un attaquant potentiel. L'attaquant a le contrôle d'un des côtés d'une interaction entre deux parties. Cette interaction peut être une communication client-serveur, entre plusieurs clients, ou plus largement le fait que l'utilisateur utilise un autre programme non prévu par le développeur.

La majorité des appareils électroniques sous le contrôle de l'utilisateur final et des programmes qu'ils exécutent peuvent être la cible d'une attaque MATE. Ces attaques peuvent cibler les logiciels utilisés sur des ordinateurs personnels, les télévisions, consoles de jeux, maisons connectées et également les cartes bancaires ou les terminaux de paiement électroniques. Selon la cible, le contrôle de l'environnement d'exécution peut être plus ou moins limité, et l'accès au code du programme plus ou moins complexe. Par exemple, observer le code exécuté par un terminal de paiement électronique est plus difficile que d'observer le code exécuté par un processeur sur un ordinateur personnel.

2.2.2 Objectifs des attaques MATE

Dans le domaine de la sécurité de l'information, on définit en général trois objectifs de sécurité :

— L'intégrité de l'information, c'est-à-dire qu'une information doit être protégée des modifications et de la destruction non autorisée. Cet objectif inclut également la non-répudiation et l'authenticité d'une information.

— La confidentialité de l'information. Une information doit être protégée des accès non autorisés.

— La disponibilité des informations. Les informations doivent être accessibles fiablement et dans un délai raisonnable.

Le but d'un attaquant d'un système d'information est d'empêcher le défenseur d'atteindre un ou plusieurs de ces objectifs. Dans le cadre de la protection logicielle et des attaques MATE, les objectifs sont les mêmes (intégrité, confidentialité et disponibilité), mais les informations à protéger sont le programme (algorithmes, protocoles ...) et ses

données (clés cryptographiques, médias propriétaires ...).

Classification des attaques

Les objectifs d'un attaquant pour analyser un programme peuvent être classés en plusieurs catégories. Selon Schrittwieser [184], les attaques peuvent être classifiées en 4 catégories :

— Localiser une donnée. L'attaquant cherche à récupérer une donnée (en clair) contenue dans un programme. Cette donnée pourra par exemple être une clé cryptographique pour un système de DRM (*Digital Right Management*), des informations d'authentification ou un certificat. Par exemple, dans le cas d'un système de DRM, un attaquant pourra utiliser les clés de déchiffrement pour récupérer une version non chiffrée du média pour le revendre à d'autres clients.

— Localiser une fonctionnalité dans un programme. Un attaquant peut souhaiter identifier où se trouve une fonctionnalité précise dans un programme. Les exemples d'utilisation de ces attaques peuvent être de vérifier si un programme implémente une certaine fonctionnalité (par exemple, pour repérer si un programme est malicieux), de repérer les mécanismes de protection du programme pour éventuellement les contourner, ou de trouver l'emplacement d'un algorithme intéressant afin de l'analyser plus précisément par la suite.

— Extraire un morceau de code. L'attaquant souhaite extraire une fonctionnalité du code avec toutes ses dépendances. Cela ne nécessite pas forcément la compréhension du code de cette fonctionnalité. Par exemple, cela peut servir pour copier un algorithme plus performant d'un concurrent (il peut être utilisé comme une *black box*). Une autre utilisation possible est de contourner un système de DRM en utilisant la fonction de déchiffrement dans un client alternatif qui ne vérifie pas les autorisations de lecture du contenu.

— Comprendre le code du programme. L'attaquant cherche à comprendre tout ou une partie d'un programme, souvent dans le but de voler la propriété intellectuelle contenue dans le programme, ou dans son environnement d'exécution (des protocoles de communication, ou des interactions avec le système)

2.2.3 Moyens des attaquants

Pour parvenir à ses objectifs, un attaquant utilisera en général plusieurs techniques mêlant l'humain et des outils automatiques [53][54]. Il est donc nécessaire de connaitre les techniques disponibles pour les attaquants afin de protéger les programmes contre ces techniques utilisées simultanément. Cette partie fait une brève présentation des techniques principales qui sont disponibles à un attaquant.

Analyse statique

L'analyse statique consiste à analyser le code du programme pour trouver des informations qui seront vraies dans toutes les exécutions du programme [91][15][18]. Il existe de nombreux algorithmes d'analyse statique qui permettent de déterminer différentes informations. Pour un même type d'information, il peut aussi exister plusieurs algorithmes avec des intérêts différents et une précision différente (l'analyse statique est, en général, indécidable [129]). En effet, pour réaliser des analyses statiques il faut faire un compromis entre la précision des informations que l'on veut obtenir et les efforts à fournir (la puissance de calcul requise pour l'analyse).

Les paragraphes suivants présentent rapidement certaines analyses statiques classiques réalisables par un attaquant [158]. Il conviendra donc, pour un défenseur, de limiter l'efficacité de ces analyses lors du choix de la protection à appliquer [58].

Analyse de flot de contrôle L'analyse du flot de contrôle [176][34] consiste à créer le graphe de flot de contrôle. Il s'agit d'une représentation graphique d'une fonction, qui est décomposée en *basic blocks,* qui seront les nœuds du graphe.

Un *basic block* contient un ensemble d'instructions, et à l'exécution du programme, l'ensemble de ces instructions sera exécuté, de la première à la dernière. Après l'exécution de la dernière instruction, le programme pourra continuer son exécution vers un autre bloc parmi un choix de plusieurs branches.

Le graphe de flot de contrôle est composé des blocs qui sont liés par les branches qui indiquent les chemins possibles d'exécutions possibles de la fonction.

Prenons par exemple le programme suivant, dont le graphe de flot de contrôle est représenté en fig. 2.1. :

```
int foo()
{
  int x = 0;
  while (x < 100)
  {
      int y = x % 2;
      if (y == 0)
        x = x * 2;
      else
        x = x + 2;
      x = x + 1;
  }
  return x;
}
```

Le graphe de flot de contrôle est une information utile à un attaquant à la fois pour une interprétation par un humain ou par des outils automatiques. Il permet d'identifier la structure d'une fonction (boucles, instructions conditionnelles ...) pour permettre une meilleure compréhension du programme.

Cette analyse est aussi utile au défenseur, en effet beaucoup de transformations vont chercher à modifier la structure du programme et à modifier le flot de contrôle. Pour s'assurer que la transformation ne modifie pas la sémantique du programme, il sera souvent nécessaire de calculer le graphe de flot de contrôle.

Certaines transformations, comme le *self-modifying code* (voir 2.5.5), vont faire que le graphe de flot de contrôle ne correspond pas forcément à l'exécution réelle. L'attaquant devra donc potentiellement développer de nouveaux outils et de nouveaux modèles [6] pour représenter la structure d'une fonction.

Pour observer la structure du programme, on peut également réaliser le même type d'analyse au niveau des fonctions. Cela sera généralement représenté par un graphe d'appel (*call graph*), où les nœuds seront les fonctions et les liens les appels de fonctions possibles. Le graphe d'appel permet lui aussi d'analyser la structure d'un programme, mais du point de vue des interactions entre les fonctions [138]. Par exemple, si un programme utilise des techniques de *tamperproofing*, c'est-à-dire des techniques ayant pour objectif d'empêcher un attaquant de modifier le programme (voir 2.3.3), et qu'on repère qu'une fonction est appelée par beaucoup de fonctions du programme qui ne font pas du tout la même chose dans le programme, on peut supposer qu'il s'agit d'une fonction de vérification de

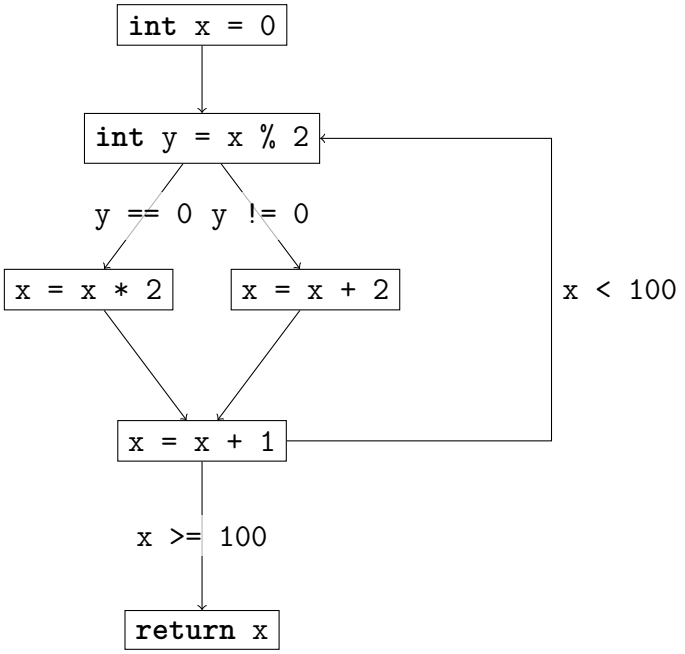

FIGURE 2.1 – Exemple de graphe de flot de contrôle(CFG)

l'intégrité du code et l'analyser plus en détail. De même, si on a repéré qu'une fonction du programme était une fonction de hachage, il y a beaucoup de chance que les fonctions qui l'appellent soient des fonctions cryptographiques.

Notons que nous présentons ici des exemples de graphes de flot de contrôle en C pour des raisons pédagogiques. Dans le cadre des attaques MATE, un attaquant n'aura accès qu'au code exécutable et devra donc réaliser cette analyse sur ce code exécutable, ce qui peut poser des difficultés supplémentaires, mais reste néanmoins possible [101].

Analyse de flux de données L'analyse de flux de données [200] a pour but de déterminer des informations sur les variables utilisées dans un programme. L'objectif de cette analyse va être de savoir à quels endroits du programme une variable est utilisée ou si elle a une valeur constante à un certain point du programme.

Dans le contexte d'une attaque MATE, cela peut permettre de trouver des constantes opaques, de contourner des mécanismes de *tamperproofing* qui se basent sur les valeurs de certaines variables, et plus généralement de mieux comprendre le programme en étant capable de suivre l'évolution des variables [197].

Analyse d'aliasing Lorsque deux pointeurs pointent vers le même emplacement en mémoire, on dit qu'ils sont alias l'un de l'autre. Dans le contexte d'une attaque MATE, et plus généralement dans un compilateur, il est utile de trouver les alias. Pour trouver ces alias, on utilisera des analyses d'aliasing [173][82][83]. Cela permet à un optimiseur de ne pas supprimer du code qui peut sembler mort, mais qui avec les alias est en réalité du code utile. De la même façon, certaines transformations d'obfuscation utilisent des alias pour cacher des informations sur l'exécution du programme. Par exemple, le *Control Flow Graph Flattening* (cf. 2.5.2) peut utiliser de l'aliasing pour cacher la structure du programme, et demander à l'attaquant de résoudre un problème complexe d'aliasing pour reconstituer la structure originale du programme.

Analyse dynamique

L'analyse dynamique [17][33] consiste à exécuter un programme et à observer certaines de ses propriétés. Contrairement à l'analyse statique, l'analyse dynamique va observer le comportement du programme avec une entrée précise et les propriétés trouvées ne seront valables que pour cette exécution avec ces paramètres d'entrée.

Les paragraphes suivants présentent les techniques d'analyse dynamique usuelles et leurs intérêts dans le cadre des attaques MATE.

Debugging Le débogueur est un outil très utile pour un développeur pour observer le comportement de son programme et résoudre des bugs. Un débogueur permet de placer des points d'arrêt, matériels ou logiciels, d'avancer pas à pas dans l'exécution et d'examiner et de modifier la mémoire du programme. Il sera donc également très utile à un attaquant [115][64].

Émulation L'émulation est un procédé qui permet d'exécuter un programme sur une implémentation logicielle d'une plateforme matérielle. Cela permet d'exécuter des programmes prévus pour être exécutés sur un matériel particulier sur un ordinateur quelconque. Cela donne accès à de nombreux outils supplémentaires pour analyser le programme [126][102][120] :

— l'attaquant peut construire une trace d'exécution et observer la mémoire du programme.
— l'attaquant peut observer le comportement complet du système d'exploitation en plus du programme et cela facilite l'analyse de programme utilisant plusieurs pro-

cessus qui coopèrent.
— cela permet de faire ces analyses sans modifier le programme, ce qui contourne certaines techniques de *tamperproofing*.

Profiling et tracing Le profilage d'un programme consiste à établir un profil d'un programme, c'est-à-dire combien de fois certaines fonctions (ou blocs) du programme sont utilisées, ou combien de temps elles prennent à s'exécuter. Pour un développeur, cela permet de détecter des bugs (par exemple, deux fonctions sont toujours appelées ensembles donc elles devraient s'exécuter le même nombre de fois, mais ce n'est pas le cas) ou de déterminer les points chauds du programme qu'il est nécessaire d'optimiser. Le *tracing* collecte la trace d'une exécution d'un programme c'est-à-dire quelles sont les fonctions (ou blocs) qui sont appelées et par qui elles ont été appelées. Un attaquant analysera souvent ces traces hors ligne, en les regroupant ou en les compressant [131]. Cela sera utile pour, par exemple, éliminer les branches qui ne semblent jamais exécutées par le programme et qui sont peut-être de fausses branches placées pour ralentir une attaque (cf 2.5.1).

Exécution symbolique

L'exécution symbolique [122] de programme consiste à simuler l'exécution d'un programme en remplaçant les paramètres d'entrée par des « symboles ». Cela permet d'observer les chemins possibles dans le programme selon le paramètre d'entrée et de calculer les valeurs d'entrée d'un programme nécessaires pour que le programme parcoure un certain chemin. Le calcul de ce chemin peut être très coûteux en ressources, en particulier sur des programmes obfusqués [19]. Certaines techniques utilisant de l'exécution dynamique en plus de l'exécution symbolique ont été développées pour améliorer son efficacité [179][187]. Ces techniques sont appelées exécution concolique (ou *Dynamic Symbolic Execution* (DSE) en anglais) et sont très efficaces contre certaines protections usuelles [165].

2.2.4 Conclusion

Nous avons présenté le principe des attaques MATE (*Man-At-The-End*) et les objectifs des attaquants. Nous avons également décrit les principales techniques et les principaux outils à disposition d'un attaquant. Ces techniques sont diverses et utilisent des vecteurs d'attaque différents. La défense du programme devra donc couvrir ces différents vecteurs

d'attaque et ces différentes techniques pour être efficace.

2.3 Défense contre les attaques MATE

2.3.1 Introduction

Pour se défendre contre les attaques MATE, il est possible d'utiliser des techniques de protection de logiciel. L'objectif de ces techniques de protection est de rendre une attaque difficile ou idéalement impossible tout en conservant les fonctionnalités et l'intérêt du logiciel. Cela signifie que ces protections ne doivent pas apporter trop de contraintes à l'utilisateur : baisse de performance, besoin de matériels supplémentaire, connexion à internet obligatoire, blocage de certaines fonctionnalités, etc. De même, elles ne doivent pas apporter trop de contraintes au développeur : coût de développement, support pour le produit, logistique de développement et de déploiement... Il sera donc souvent nécessaire de faire des compromis entre l'efficacité de la protection et l'impact de ces protections sur les développeurs et les utilisateurs [46].

Nous allons présenter plusieurs techniques de protection de logiciels contre les attaques MATE : l'obfuscation, le *tamperproofing*, le *watermarking* et la diversification artificielle de programme. Nous verrons également comment ces techniques peuvent être combinées pour améliorer la protection du programme.

2.3.2 Obfuscation

Une définition possible de l'obfuscation de programme est le fait de modifier un programme pour le rendre plus difficile à comprendre que la version originale [161][72] tout en conservant les mêmes fonctionnalités que le programme original. Cette difficulté peut être quantifiée par divers concepts : le temps humain pour comprendre un programme, la puissance de calculs nécessaires pour un outil automatique ou encore les moyens financiers nécessaires. Il est souvent difficile de quantifier cette valeur, des pistes sont données en 2.7.

L'obfuscation de programme n'est pas simplement de la sécurité par l'obscurité (*security through obscurity*), un terme souvent utilisé en cryptographie. Il désigne les algorithmes qui sont censés rester secrets, alors que le consensus est que les algorithmes doivent être publics afin qu'un maximum de personnes puisse les étudier afin d'éliminer les failles de sécurité potentielle. Dans le domaine de l'obfuscation, on considère généralement que

les transformations existantes sont publiques [161]. Le procédé d'application est lui généralement secret : quelles transformations sont appliquées ? Dans quel ordre sont-elles appliquées ? Quels paramètres sont utilisés pour l'exécution des transformations ?

Il est possible de transformer un programme à la main pour l'obfusquer, mais ce procédé est long, avec des risques d'erreurs et nécessite une expertise en obfuscation. En général, le processus d'obfuscation de programme sera réalisé par un outil automatique appliquant un ensemble de transformations contenues dans une bibliothèque de transformations. Ce programme est appelé un obfuscateur.

Un obfuscateur automatique est un programme prenant en entrée un programme non protégé et un ensemble de paramètres et qui donne en sortie un programme obfusqué. Selon l'obfuscateur, il peut exister plusieurs niveaux d'abstraction pour les paramètres donnés à l'obfuscateur. Idéalement, un obfuscateur destiné à des utilisateurs non experts en obfuscation prendra en paramètre le niveau d'obfuscation souhaité et les parties du programme à protéger en priorité. Un obfuscateur destiné à des utilisateurs experts pourra prendre un « script » d'application des transformations pour paramétrer l'application des transformations [45].

2.3.3 Tamperproofing

Lorsqu'un développeur de programme veut empêcher un attaquant de modifier son programme, il peut utiliser des techniques de *tamperproofing*. Le *tamperproofing* consiste à générer des effets de bord sur le programme lorsque celui-ci est modifié (ou observé). Il peut s'agir de faire « planter » le programme, de faire des modifications sur le système de l'attaquant ou encore d'envoyer un message signalant que le programme est en train d'être attaqué.

La protection de *tamperproofing* peut se décomposer en deux parties : la détection de l'attaque et la réponse à l'attaque.

La modification d'un programme dans le contexte des attaques MATE, cherche en général à provoquer des comportements non prévus dans le programme, comme l'utilisation d'un débogueur ou le contournement de vérifications. La détection de l'attaque cherchera donc à détecter les comportements non prévus du programme [38]. Les techniques usuelles de détection d'attaque sont le calcul de *checksum* sur le code du programme avec une comparaison avec la valeur prévue. Une autre possibilité est de vérifier que le programme est dans un état d'exécution possible en observant les variables en mémoire [193].

Une fois l'attaque détectée, le programme doit fournir une réponse. On essaiera en

général de faire que cette réponse ne se déclenche pas au moment où la modification est détectée, mais à un moment qui apparait aléatoire de l'exécution du programme afin de complexifier la tâche de contournement du *tamperproofing*.

L'action de lire son propre code n'est en général pas un comportement normal d'un programme, il est souvent simple pour un attaquant de détecter les fonctions de *tamperproofing* contenues dans un programme, et donc de les contourner. Il pourra par exemple remplacer la fonction de détection par une fonction qui répond que le code n'est jamais modifié. Ces fonctions doivent donc être également protégées, soit par de l'obfuscation (cf 2.3.6), soit par du *tamperproofing* appliqué sur elles-mêmes[13].

Le but du tamperproofing peut également être de détecter un environnement d'exécution « dangereux » [166][114], c'est-à-dire un environnement dans lequel le programme n'est pas censé être exécuté, car cet environnement permet d'observer plus facilement la mémoire du programme ou ses interactions avec le système par exemple. Cela est en particulier utilisé par les *malwares* [137][49], qui veulent avoir un comportement qui semble bénin quand ils sont exécutés sur une machine virtuelle ou dans un bac à sable (*sandbox*) afin d'éviter la détection, et ne révéler leur nature néfaste que sur le système cible.

2.3.4 Watermarking

Dans le domaine de la protection de médias, il existe de nombreux travaux pour insérer des identifiants uniques dans des médias numériques tels que des images, du son, des vidéos ou des textes [185][172][81]. Il y a deux types de marques (*watermark* en anglais). La première est un identifiant de client qui est unique à chaque personne à laquelle le produit a été distribué. La seconde marque est un *copyright notice*, qui est identique sur toutes les reproductions du média et qui indique le propriétaire du média.

Si une copie illégale du média est trouvée, le propriétaire peut utiliser l'identifiant unique pour trouver la personne ayant copié ou distribué sa copie et engager des poursuites judiciaires. Si une autre personne prétend qu'il est l'auteur de ce média, l'extraction du *copyright notice* permet de prouver la paternité du média.

Le *watermarking* de média utilise en général les limites des sens humains, en modifiant de façon imperceptible par un humain le média, tout en restant détectable par un algorithme. Par exemple, en ajoutant des fréquences inaudibles dans un son, ou en changeant légèrement la couleur de quelques pixels d'une image. De plus, il est nécessaire que ces marques résistent au bruit. En effet, il ne faut pas qu'un attaquant puisse dégrader légèrement la qualité du média et rende la marque indétectable. Pour cela un attaquant

peut, par exemple, prendre en photo une image ou la compresser.

Pour la protection de logiciel, les objectifs sont similaires : prouver la paternité du logiciel ou détecter la source de copies illégales.

Il existe deux types de marques dans un programme. Les marques statiques qui sont contenues directement dans le code (ou son binaire, ou sa représentation intermédiaire) du programme. Par exemple, Moskowitz [159] propose d'intégrer un contenu multimédia (comme une image) dans la section des données statiques d'un programme. Ce contenu est marqué par les méthodes usuelles de *watermarking* des contenus multimédias.

Les marques dynamiques [71] ne sont visibles qu'à l'exécution d'un programme, en général lorsqu'une entrée secrète est donnée au programme. Par exemple, l'algorithme de Collberg-Thombornson (CT) [68], construit la marque durant l'exécution grâce à une clé secrète. Cette clé secrète est une séquence d'entrée du programme, lorsque le programme est exécuté avec ces entrées, il prend un chemin d'exécution particulier et génère la marque.

En général, on utilisera plusieurs marques insérées avec différentes méthodes pour rendre la tâche d'un attaquant plus difficile [76]. En effet, il peut en retirer une, puis distribuer le programme, sans se rendre compte qu'il y en a toujours d'autres dans le programme.

2.3.5 Diversité de programme

Pour améliorer l'efficacité du développement logiciel et réaliser des économies d'échelle, les systèmes informatiques sont de plus en plus homogènes, qu'il s'agisse du matériel, des systèmes d'exploitation, des environnements d'exécution (interpréteur, machine virtuelle)... Une application peut être développée une seule fois et être exécutée sur un ordinateur personnel, un téléphone, une télévision et via une interface Web et cela chez tous les clients, sans coût supplémentaire pour le développeur.

Malheureusement, cette homogénéité a aussi un effet négatif sur la sécurité des programmes [99]. Un attaquant peut télécharger une version du logiciel, trouver une faille de sécurité dans celle-ci et l'appliquer telle quelle pour attaquer tous les autres possesseurs du logiciel.

Le type d'attaque pouvant profiter de cette homogénéité est divers. Il peut s'agir d'attaque permettant d'exploiter des failles de sécurité pour récupérer des informations (clés cryptographiques privées) ou prendre le contrôle d'un système faisant tourner ce logiciel. Lorsqu'une telle faille est découverte, de très nombreux systèmes se retrouvent vulnérables à une unique attaque. Par exemple, dans le cas de Heartbleed [154], près de la

moitié des sites web offrant une connexion chiffrée (*https*) du monde ont été affectés [86].

Le cas des attaques MATE est similaire, en effet si un attaquant découvre une faille permettant de contourner un système de *DRM* ou de licence, il pourra la diffuser ou la vendre aux autres utilisateurs. L'homogénéité des programmes n'est pas forcément un mauvais point pour d'autres objectifs des attaques MATE. Pour voler la propriété intellectuelle d'un algorithme, il suffit de comprendre le code d'une seule version du programme, même s'il en existe plusieurs. Au contraire, s'il existe plusieurs versions, il y en a potentiellement une plus facile à attaquer que les autres.

Dans tous les cas, les attaques sur les logiciels ont très souvent un intérêt économique [4]. Un attaquant peut vendre les informations sur une faille de sécurité, ou un « crack » permettant de contourner un système de licence. Une attaque peut aussi être utilisée pour infliger des pertes économiques à une entité, en empêchant l'accès à ses services ou en lui infligeant des pertes économiques en rendant ses produits disponibles gratuitement.

La diversification de programme consiste à distribuer un programme dans plusieurs versions, fonctionnellement identiques, mais exécutant un code différent [116]. L'intérêt de la diversification est donc économique : s' il existe plusieurs versions, une attaque sur une version ne marchera que sur cette version, et non pas sur l'ensemble des programmes. L'attaquant devra donc soit investir plus de moyens pour faire fonctionner l'attaque sur toutes les versions, soit se contenter d'un impact réduit pour son attaque. La rentabilité de l'attaque (l'impact sur le coût de l'attaque) diminue donc fortement avec la diversification de programmes. De façon similaire, on peut également ajouter de la diversité dans l'exécution des programmes, en utilisant par exemple une protection de *self modifying code* 2.5.5. Cette diversité à l'exécution augmente également le coût des attaques réalisées à l'exécution comme l'analyse de traces.

On pourrait développer plusieurs versions d'un logiciel manuellement, mais cela sera souvent très coûteux. En général, on essaiera d'ajouter de la diversité artificielle avec un outil automatique pendant le processus de compilation [130]. Le processus de diversification artificiel est similaire aux autres techniques de protection logicielle : un outil automatique va appliquer un ensemble de transformations (choisies aléatoirement, ou ayant un comportement dépendant d'une graine aléatoire) sur un programme.

2.3.6 Utilisation combinée de ces techniques

Même si l'obfuscation, le *tamperproofing*, le *watermarking* et la diversification artificielle sont des techniques de protection avec des objectifs différents et ciblant des types d'attaques différentes, elles seront souvent combinées.

Lorsqu'un attaquant va vouloir modifier un programme protégé par du *tamperproofing*, son objectif va être de trouver les fonctions de détection ou de réponses du programme et de les contourner. Un des meilleurs moyens pour le défenseur de protéger ces fonctions est d'utiliser de l'obfuscation [79] [170]. Ainsi, l'attaquant devra d'abord comprendre le programme et son obfuscation pour enlever ou contourner le *tamperproofing*.

De la même façon, une *watermark* doit être cachée pour qu'elle soit difficile à être détectée et retirée. Il sera donc souvent intéressant de protéger sa *watermark* avec de l'obfuscation pour la cacher, et du *tamperproofing* [110][13] pour empêcher de la retirer. Cependant certaines *watermark*, notamment beaucoup de *watermark* statiques, se basent sur la forme du code, ce qui peut être cassé par certaines transformations. En effet, cette protection, et de nombreuses autres protections ne conservent pas la sémantique du programme, il est donc nécessaire d'être attentif à la compatibilité entre ces protections (voir 3.3.3).

Dans le cadre de l'obfuscation, un attaquant va utiliser de nombreux outils. Dans le cadre d'une attaque dynamique, certains de ces outils, comme les débogueurs, fonctionnent en modifiant le code du programme. Il peut donc être très efficace d'utiliser des techniques de *tamperproofing* pour rendre la tâche de rétro-ingénierie plus difficile, en limitant les outils disponibles pour l'attaquant [61][148].

La composition de ces différentes protections, et de plusieurs techniques de chacune de ses catégories de protection peut donc augmenter significativement le niveau de sécurité de ces applications à condition que cette composition soit correctement effectuée [1].

2.3.7 Impact sur les performances

Les techniques de protection de logiciel ont un impact significatif sur les performances d'un programme. Intuitivement, la plupart des protections vont affecter négativement les performances (vitesses d'exécution et/ou mémoire). Elles font des opérations très coûteuses : elles remplacent des constantes par des calculs complexes, font exploser le nombre de branches du graphe de flot de contrôle pour empêcher l'application de certaines analyses statiques, chiffrent les données, etc.

	`-O2`	`-bcf -fla -sub_loop=3 -O2`
taille	96 KiB	1.11 MiB
temps d'exécution	3.77 s	50.63 s

FIGURE 2.2 – Comparaison de la taille de binaire et de la vitesse d'exécution sur un fichier de 100 MiB du programme gzip obfusqué (`-bcf -fla -sub_loop=3 -O2`) et non obfusqué(`-O2`)

L'impact sur les performances est souvent significatif. Par exemple, nous avons mesuré l'impact de l'utilisation de `obfuscator-llvm` [119] sur le programme `gzip` [95] en le compilant avec `Clang` [134]. Nous avons comparé les performances du programme entre la version obfusqué (avec la substitution d'instruction, la création de faux flots de contrôle (cf. 2.5.1) et le *control flow flattening*) et une version compilée directement avec `Clang`. Dans les deux cas, le programme a été compilé avec les optimisations usuelles (`-O2`).

Les performances des deux programmes ont ensuite été testées sur un fichier de 100 Mo (fig. 2.2). Nous avons également regardé la taille du binaire de ces programmes.

La taille du binaire est multiplié par plus de 100, ce qui peut être significatif dans certains cas, notamment pour les systèmes embarqués. De plus, le temps d'exécution est lui multiplié par 13, ce qui sera inacceptable dans beaucoup d'applications.

Il s'agit ici d'une simple illustration de l'impact sur les performances des transformations d'obfuscation et pas d'une étude complète sur le sujet. Elle montre tout de même qu'il est nécessaire de prendre en compte les performances dans la création de transformations d'obfuscation et la composition de ces transformations.

2.3.8 Conclusion

Nous avons présenté les différentes techniques de protection de logiciel qui permettent de renforcer la sécurité des programmes contre les attaques MATE. Ces techniques peuvent être utilisées individuellement ou combinées pour offrir une protection plus résistante et protéger contre plusieurs vecteurs d'attaques. Cependant, il est nécessaire de prendre en compte l'impact sur les performances de ces techniques lors de leur application sur un programme. Un développeur devra parfois faire un compromis entre les performances du programme et le niveau de protection apportés par ces techniques.

2.4 Obfuscation cryptographique

Nous allons présenter brièvement ici certaines définitions théoriques et cryptographiques de l'obfuscation, et montrer pourquoi elles ne sont pas pour le moment utilisables en pratique. Notons que l'utilisation du terme « cryptographique » ici ne signifie pas que nous utilisons des techniques de chiffrement sur le code source des programmes, mais plutôt que l'obfuscation, de façon prouvable, ne laisse pas fuiter d'informations sur le programme (tout comme le chiffrement d'un message ne laisse pas fuiter d'informations sur le message). Les informations qui ne doivent pas fuiter dépendent de la définition formelle de l'obfuscation cryptographique. Nous allons ici présenter les deux définitions les plus courantes de l'obfuscation cryptographique.

Les transformations d'obfuscation cryptographique sont majoritairement représentées sur des circuits booléens plutôt que directement sur des programmes. Ces circuits peuvent être traduits en programmes, mais sont moins expressifs, ce qui facilite les preuves sur ces transformations.

2.4.1 Obfuscation boite noire virtuelle

Un algorithme $\mathcal{O}$ est un obfuscateur créant des boites noires virtuelles [25] si :

— Pour tous les circuits C, le circuit $\mathcal{O}(C)$ réalise la même fonction que C (l'obfuscateur préserve la sémantique).

— La durée d'exécution et la taille du circuit $\mathcal{O}(C)$ sont au pire polynomialement plus grandes que C.

— On ne peut pas extraire plus de propriétés sur le circuit $\mathcal{O}(C)$ que sur un oracle de C (la connaissance seulement des entrées sorties).

Un obfuscateur qui respecterait cette définition permettrait de créer des programmes qui ne donnent aucune information sur leurs fonctionnements internes, mais réalisent toujours leurs objectifs dans un temps « raisonnable ». Le programme obfusqué sera alors une simple boite noire. Autrement dit, l'accès au programme ne donnera pas plus d'information à un attaquant que la simple observation des entrées et des sorties du programme.

Barak et al. [25] ont prouvé qu'un obfuscateur boite noire virtuelle était impossible avec cette définition. Ils démontrent aussi que l'obfuscateur boite noire virtuelle est également impossible même en affaiblissant la définition [26] sur le temps d'exécution ou la préservation approximative de la sémantique.

2.4.2 Obfuscation indistinguable

L'obfuscation indistinguable est une autre définition d'une obfuscation cryptographique. Étant donné deux circuits équivalents (c'est-à-dire avec les mêmes entrées/sorties) et leurs versions obfusqués, un attaquant ne peut pas distinguer avec une probabilité non négligeable quel circuit obfusqué vient de quel circuit original [97]. L'obfuscation indistinguable permet d'atteindre la *best possible obfuscation* [100]. Un attaquant ne peut pas le distinguer du programme le mieux obfusqué ayant la même fonctionnalité et la même taille.

L'application de l'obfuscation indistinguable est encore loin d'être pratique : le temps de compilation, d'exécution et la mémoire occupée par un circuit indistinguable produit par les implémentations actuelles sont beaucoup trop élevés. Par exemple, une implémentation d'une multiplication sur 2 bits, protégée par de l'obfuscation indistinguable, compilée sur un ordinateur moyen nécessite 10^{27} années et 20 Zetta octets de mémoire pour la compilation et 1.3×10^8 années pour l'exécution [21]. La protection de cette implémentation est obtenue grâce à une difficulté algébrique et est basée sur une variation des applications multilinéaires : les *Multilinear Jigsaw Puzzles* [97]. Ce domaine de recherche est devenu très actif récemment et a vu beaucoup de progrès. De nouvelles propositions d'implémentations [36][35] utilisant le chiffrement fonctionnel [41][141] avec de bien meilleures performances avec une sécurité possiblement légèrement réduite. Il existe également des propositions de méthodes qui se limitent à la protection d'un sous-ensemble des programmes, plutôt qu'un programme quelconque, qui produisent des résultats intéressants [32][124].

2.4.3 Conclusion

Bien qu'offrant une protection intéressante, efficace et mesurable, l'obfuscation cryptographique n'est, pour le moment, pas utilisable en pratique. En effet, l'obfuscation boite noire virtuelle est impossible et les techniques actuelles d'obfuscation indistinguables ont un impact trop élevé sur les performances et ses utilisations, bien qu'intéressantes, sont limitées. En pratique, un défenseur devra donc utiliser d'autres techniques de protection, moins formelles, qui sont présentées dans la suite de ce chapitre.

2.5 Transformations courantes

Comme l'obfuscation cryptographique n'est pour l'instant pas utilisable en pratique, d'autres solutions sont employées pour protéger les programmes. Nous allons présenter quelques transformations de code qui sont souvent utilisées dans la pratique [16]. Ces transformations sont souvent combinées pour offrir une meilleure protection.

2.5.1 Flot de contrôle « menteur »

L'objectif de cette transformation est de rendre le flot de contrôle plus difficile à comprendre en le rendant « faux » ou « menteur » (*bogus control flow*). Pour cela, on peut ajouter des branches qui ne seront jamais empruntées, des branches qui semblent être exécutées que dans certains cas, mais qui seront toujours exécutées en réalité, des branches qui vont parfois être prises et parfois non, mais sans que cela modifie le résultat du programme [72][69].

La résistance de cette protection aux attaques va principalement être liée à la résistance des prédicats opaques qui choisissent quelles branches vont être prises. En effet, si un attaquant détecte qu'un prédicat est toujours vrai (ou faux), il pourra simplement retirer les fausses branches. Certaines techniques pour créer des prédicats opaques sont présentées en 2.5.3.

Un autre point potentiel d'attaque est l'analyse des blocs présents dans les fausses branches. Si ces blocs sont vides, qu'ils n'affectent en rien le déroulement du programme ou possèdent un comportement qui apparait immédiatement aléatoire et n'ayant aucun sens, un attaquant pour déduire que les branches qui les contiennent sont fausses. Enfin, cette protection est vulnérable aux analyses dynamiques, notamment les analyses de traces, car les branches jamais exécutées en pratique pourront être ignorées par l'attaquant. Si l'ensemble du programme est obfusqué (y compris les fausses branches) et que les « faux blocs » sont générés de façon intelligente (par exemple en les extrayant d'autres endroits du programme ou utilisant des variables du contexte), cela sera beaucoup plus difficile pour un attaquant.

2.5.2 Aplatissement du graphe de flot de contrôle

Les transformations de *Control Flow Graph Flattening* [48] détruisent la structure des fonctions créée par les structures conditionnelles et les niveaux de boucles. Elles « appla-

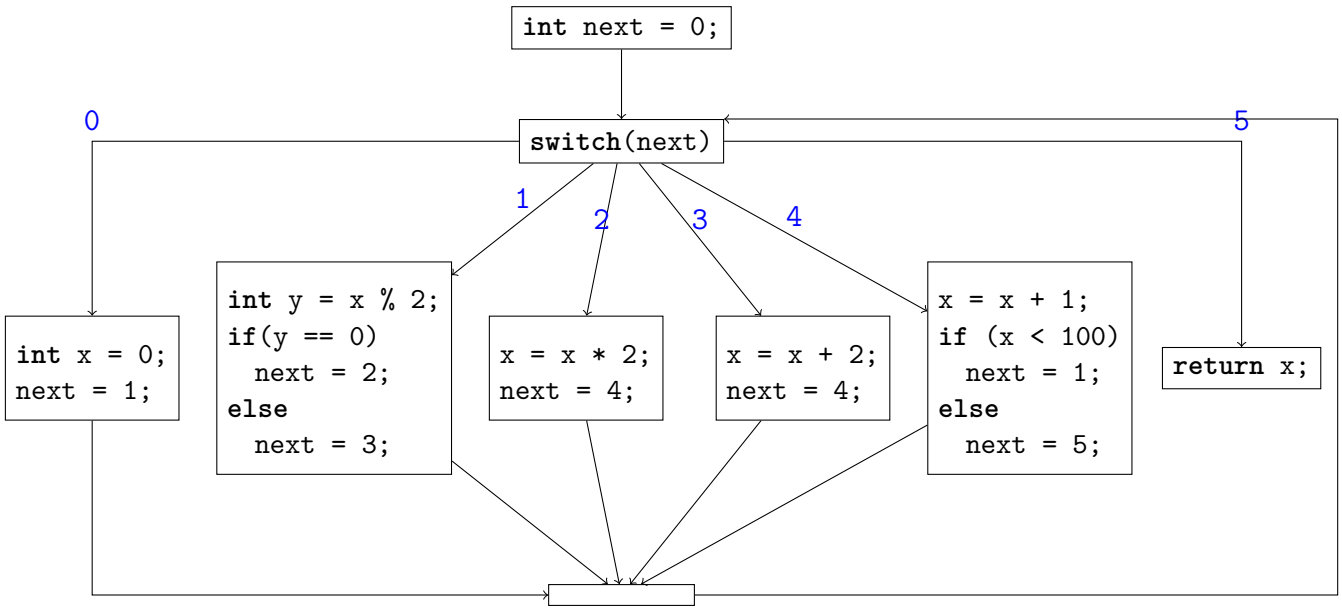

FIGURE 2.3 – Exemple de CFG Flattening

tissent » le graphe de flot de contrôle en plaçant tous les *basics blocks* du programme dans une instruction *switch* à l'intérieur d'une boucle infinie [72][132]. Chaque bloc indique son successeur à l'aide d'une variable *next*. Cette variable étant la clé permettant de comprendre et de reconstituer la structure de la fonction, il est important de la protéger en utilisant d'autres techniques d'obfuscation.

Par exemple, le graphe de flot de contrôle du programme présenté en fig. 2.1 peut être aplati pour devenir le graphe en fig. 2.3. Sur cette représentation graphique, on remarque que la structure du programme est complètement horizontale (d'où le nom de la transformation). Un attaquant mettra probablement plus de temps à comprendre cette structure plutôt que celle du programme initial.

Cette transformation sera plus efficace (d'un point de vue de la protection) si le nombre de blocs composant la fonction est élevé. En effet, plus il y a de blocs dans la fonction, plus elle sera « applatie ».

2.5.3 Prédicats opaques

Plusieurs transformations d'obfuscation, par exemple le *bogus control flow* (2.5.1) et le *Control Flow Graph Flattening* (2.5.2), utilise des prédicats pour contrôler le flux d'exécution des programmes. Par exemple, les valeurs associées à la variable *next* dans la figure 2.3 représentent de tels prédicats. La sécurité apportée par la protection repose, au moins en partie, sur la difficulté pour un attaquant à identifier et comprendre ces prédicats. La valeur de ces prédicats est constante en un point du programme (en tout cas dans une exécution nominale du programme) est connue du développeur, mais doit donc être cachée à l'attaquant. Le but des algorithmes de génération de prédicats opaques est de générer des prédicats qui sont difficiles à résoudre pour un humain et un algorithme [204].

Pour cela on peut utiliser des problèmes mathématiques (notamment reposant sur la divisibilité). Mais il est difficile de générer ce genre d'équation à la volée et si on utilise une bibliothèque d'équation, on peut supposer qu'un attaquant aura accès à cette même bibliothèque. Par exemple, on sait que $\forall x \in \mathbb{Z}, (x^2 + x)\%2 = 0$, on pourra générer le prédicat opaque simple `(x * x + x) % 2 == 0` qui sera toujours vrai (si $x \in \mathbb{Z}$).

Le défenseur peut également créer des prédicats basés sur de l'*aliasing* qui comparent des pointeurs [73]. Pour casser ces prédicats opaques, l'attaquant devra résoudre un problème d'*aliasing* difficile. Le défenseur peut notamment créer des prédicats opaques qui s'attaquent aux faiblesses des outils d'analyses statiques, par exemple l'analyse de calculs parallèles, ou l'utilisation d'exceptions ou d'interruptions [56][112][109][85].

Le développeur peut également rendre le calcul de ces prédicats plus difficile en transformant les opérations mathématiques traditionnelles (addition, soustraction, division, multiplication) en *Mixed Boolean Arithmetic* [203][106], c'est-à-dire en transformant ces opérations en calculs utilisant à la fois les opérateurs mathématiques et les opérateurs logiques bit à bit (AND, OR, XOR et NOT). Par exemple, l'expression $x + y$ avec $x, y \in \{0, 1\}$ calculée sur 8 bits, peut se transformer en $(((x \oplus y) + 2 \times (x \wedge y)) \times 39 + 23) \times 151 + 111$ en utilisant cette technique [92].

2.5.4 Virtualisation

L'obfuscation par la virtualisation [20][67] consiste à traduire le code du programme à protéger dans un autre langage, en général un langage généré spécifiquement pour le programme en question, et à générer l'interpréteur de langage.

Cela force l'attaquant à comprendre le langage cible avant de pouvoir comprendre le

programme exécuté par l'interpréteur. De plus, cela rend l'utilisation de certains outils, notamment d'analyses statiques, moins utiles. En effet, ces outils vont permettre de déterminer des propriétés sur le fonctionnement de l'interpréteur, mais pas sur le programme exécuté, à moins de développer des outils destinés spécifiquement à analyser le langage dans lequel le programme est écrit (ce qui est un cout important pour l'attaquant).

2.5.5 Code auto modifiant

L'objectif de ces transformations est de faire que le code « statique » du programme n'est pas celui qui sera exécuté. Pour cela, avant d'exécuter le morceau de code protégé le programme va d'abord modifier ce code dans la mémoire ou dans le cache du processeur, exécuter le nouveau code, et potentiellement remettre le code original dans la mémoire. On peut placer de fausses instructions dans le code et les remplacer à un moment de l'exécution par les vraies instructions [121]. On peut également créer un programme qui est en permanence en train de se modifier et dont seule la partie en train d'être exécutée est correcte [14]. Cette méthode peut également être utilisée par des *malware* [186] pour cacher dynamiquement du code malicieux et éviter d'être repéré par un antivirus.

Étant donné que le code exécuté par le processeur doit être le code « en clair », cette protection est assez inefficace contre les émulateurs ou les débogueurs qui peuvent prendre une image du code au moment de l'exécution pour l'analyser.

2.5.6 Autres techniques d'obfuscation de programmes

Nous avons jusqu'à présent présenté des techniques de protection basées sur des transformations de codes des programmes cibles. Dans cette section, nous présentons quelques autres techniques de protection de programmes qui ne sont pas des transformations de code et qui sont donc en dehors du cadre de cette thèse.

Chiffrement

Pour attaquer un programme, la première étape sera souvent de récupérer le code du programme. Le chiffrement vise à rendre cette étape plus difficile. Il est possible de chiffrer un programme, et de le déchiffrer uniquement à l'exécution, potentiellement en déchiffrant juste la partie du programme nécessaire plutôt que tout le programme [175]. Comme le programme doit être exécuté, celui-ci devra toujours être déchiffré à un moment et la clé de déchiffrement devra être contenue quelque part dans le programme. Cette technique

est beaucoup utilisée par les virus, afin de cacher leurs signatures et empêcher les antivirus de les analyser statiquement.

Protections matérielles

En général, il sera plus simple d'attaquer un programme s'exécutant sur un ordinateur traditionnel que sur du matériel spécifique. En effet, de nombreux outils (débogueur, désassembleur, outils d'analyses...) sont bien développés pour les matériels standards.

Un programme exécuté sur un matériel spécifique sera plus couteux à attaquer, l'attaquant devra d'abord comprendre le fonctionnement du matériel et aura potentiellement besoin d'outils moins accessibles qu'un simple PC (laser pour modifier le code exécuté [191], oscilloscope pour mesurer la consommation ou les champs électromagnétiques [202]).

De plus, il est possible d'intégrer des protections directement dans le matériel : on peut distribuer un programme chiffré avec la clé de déchiffrement contenu dans le processeur [108]. Ainsi, le code déchiffré n'est disponible qu'à l'intérieur du processeur. On peut également utiliser des clés cryptographiques pour authentifier le matériel et empêcher le programme de s'exécuter s'il ne s'exécute pas sur la plateforme prévue.

Le principal défaut de la protection matérielle est qu'elle est couteuse, pour le développeur qui doit développer et produire la plateforme matérielle en plus du programme, et pour l'utilisateur qui ne peut se servir du matériel qu'il possède déjà.

2.5.7 Conclusion

Dans cette partie, nous avons présenté plusieurs transformations logicielles pour l'obfuscation de programme qui sont représentatives des solutions utilisées en pratique pour la protection de programmes. Nous avons également présenté d'autres types de protections existantes, qui peuvent être intéressantes, mais ont des cas d'applications plus limités. Dans la suite de cette thèse, nous nous concentrerons sur les protections logicielles.

2.6 Classification des transformations

Les transformations pour l'obfuscation sont diverses dans leurs exécutions et leurs objectifs. Elles peuvent néanmoins être classifiées selon plusieurs critères [72][146][130][23] :

Transformation	Moment d'application	Unité de code ciblé	Statique/dynamique	Cible
Flot de contrôle menteur	Compilation	Flot de contrôle	Statique	Algorithme
Aplatissement du CFG	Compilation	Flot de contrôle	Statique	Algorithme
Prédicat opaque	Compilation	Valeur	Statique	Algorithme/Données
Virtualisation	Compilation	Quelconque	Dynamique	Algorithme
Code auto modifiant	Exécution	Instruction	Dynamique	Algorithme/Données
Chiffrement	Compilation/Exécution	Quelconque	Dynamique	Algorithme/Données
Protections matérielles	Exécution	Quelconque	Statique/Dynamique	Algorithme/Données

TABLE 2.1 – Classification de quelques transformations courantes

— Le niveau d'abstraction du programme sur lequel est appliquée la transformation (source, binaire, représentations intermédiaires). Une transformation peut être plus simple à réaliser pour un niveau d'abstraction que sur un autre, et l'outil réalisant ces transformations dépendra du niveau d'abstraction.
— Le moment auquel la transformation est exécutée : celle-ci peut être appliquée directement à l'implémentation, pendant la compilation ou encore quand le programme est exécuté.
— L'unité de code qui est modifiée par la transformation (instructions, boucles, fonctions . . .).
— Une transformation d'obfuscation peut être statique ou dynamique. Une transformation statique est appliquée avant l'exécution du programme, et le programme ne change pas durant son exécution. Pour une transformation dynamique, le programme change durant son exécution.
— La cible que la transformation souhaite protéger : les données ou les algorithmes du programme.

Classifier les transformations selon ces critères permet de déterminer les transformations adaptées au programme que l'on souhaite protéger et aux objectifs de protection de l'utilisateur. Chaque transformation peut avoir son utilité selon le contexte, étant donné qu'il n'y a pas d'obfuscation « parfaite » (cf. 2.4.1), on doit choisir les transformations adaptées à nos besoins.

La table 2.1 donne une classification possible des transformations que nous avons décrites en 2.5. Notons que nous n'avons pas indiqué le niveau d'abstraction sur lequel est appliquée la transformation, car celui-ci dépendra de l'implémentation exacte de la transformation (nous n'avons présenté que le concept global de ces transformations). De la même façon, certains critères peuvent être différents ou plus précis que ceux qui sont présentés ici selon l'implémentation.

2.7 Mesure de l'efficacité de la protection

Lorsqu'on protège un programme avec les techniques présentées dans ce chapitre, le développeur souhaite en général connaître le niveau de protection qu'il a atteint. Étant donné qu'il n'est pas possible d'obtenir une protection « incassable » (cf. 2.4.1), et est même théoriquement « simple » à casser [8], l'information à mesurer sera « Casser la protection prendra X temps/ coutera Y argents » plutôt qu'une information binaire (« le programme est/n'est pas protégé »). Cependant mesurer cette protection est loin d'être trivial, contrairement à une mesure de la qualité d'une optimisation, il n'est pas possible de « juste » exécuter le programme et regarder la durée qu'il prend à s'exécuter. Nous présentons ici quelques critères et techniques permettant d'obtenir des informations sur la qualité d'une obfuscation.

Collberg [72] définit comment mesurer la qualité de l'obfuscation d'un programme, selon plusieurs critères :

— la *potency* qui représente la difficulté à lire le code d'un programme, cette métrique est à priori abstraite, mais peut également utiliser certains critères qualitatifs ou quantitatifs que nous présenterons ci-dessous ;
— la *resilience* qui mesure la résistance de l'obfuscation aux outils automatiques de désobfuscation : ajouter du code aléatoire qui n'est jamais exécuté augmente la *potency*, mais peut facilement être éliminée par un programme tel que l'optimiseur d'un compilateur ;
— les performances du programme obfusqué par rapport au programme original ;
— la furtivité de l'obfuscation, c'est-à-dire la difficulté pour un attaquant de détecter la partie du code qui a été protégé.

Ces paramètres peuvent servir de critères à la mesure de la qualité d'une obfuscation, mais il n'est pas toujours simple, ou même possible d'obtenir une valeur numérique pour ces critères. Il faut donc parfois utiliser d'autres métriques.

Il est très difficile de mesurer la *potency* d'une obfuscation, car il s'agit principalement d'une mesure de la difficulté de casser la protection pour un humain. Ankart et al. [7] proposent d'utiliser des métriques de complexité de programme ; telle que le nombre d'instructions, la complexité cyclomatique et la complexité du flot de données ; pour mesurer l'efficacité d'une protection. Tandis que Ceccato et al. [52][51] mesurent l'efficacité de rétro-ingénierie de sujet humain pour évaluer l'efficacité de plusieurs types de protection et Manikyam et al. [145] utilisent également des experts humains pour comparer l'efficacité

de plusieurs obfuscateurs automatiques.

Banescu [22] propose un modèle formel d'attaquant utilisant des outils automatiques qui permet d'évaluer la *resilience* d'un programme.

Ces techniques, et d'autres que nous n'avons pas présentés ici permettent d'obtenir des informations intéressantes, mais non exhaustives, sur la qualité et la performance de la protection d'un programme.

2.8 Conclusion

Dans ce chapitre nous avons présenté la sécurité des logiciels du point de vue des attaques *Man-At-The-End* (MATE), c'est-à-dire les attaques réalisées sur le programme par son utilisateur. Nous avons étudié les objectifs des attaquants, les informations qu'ils peuvent chercher à obtenir et les outils qu'ils peuvent utiliser sur le programme. Nous avons présenté les outils et autres moyens disponibles pour réaliser des attaques MATE et atteindre ces objectifs, ainsi que pour contourner les protections mises en place. Ensuite, nous avons présenté les principes de l'obfuscation cryptographiques, qui ne sont pas à l'heure actuelle utilisables en pratique ; ainsi que les principes des obfuscations réalisées en pratique. Nous avons notamment présenté plusieurs transformations d'obfuscation courantes, qui sont représentatives des obfuscations possibles, ainsi que comment ces transformations peuvent être classifiées et comment leur efficacité peut être mesurée. Ces transformations seront utilisées dans les exemples de la suite de cette thèse.

Dans les chapitres suivants, nous étudierons plus en détail le processus de protection de programme, en particulier de son obfuscation. Nous présenterons le processus de compilation et la composition de transformations d'obfuscation, nous étudierons les problématiques spécifiques à ces processus et proposerons des solutions à ces problématiques.

Chapitre 3

Chaine de compilation et d'obfuscation

3.1 Introduction

Le code d'un programme subit de nombreuses transformations entre le code source initial et le code exécutable. Une partie de ces transformations est généralement réalisée par un compilateur. Le compilateur a un rôle essentiel sur les performances du programme final, car la génération du code exécutable et les transformations d'optimisation réalisées par le compilateur ont un impact critique sur les performances d'un programme. Le choix des bonnes transformations d'optimisation peut plus que doubler la vitesse d'un programme [50] ! De même, le compilateur est un outil très important dans la sécurité d'un programme. En effet, le compilateur peut avoir un impact négatif sur la sécurité d'un programme, notamment en retirant ou affaiblissant les protections mises en place préalablement sur le programme [198][190], ce qui est souvent le cas avec les compilateurs usuels [1]. Mais, le compilateur peut également être un outil neutre pour la sécurité des programmes, en conservant les propriétés de sécurité des programmes, voire bénéfique, en augmentant la sécurité, en exécutant des transformations de sécurité lors de la compilation. Pour cela, il est nécessaire que le compilateur ait été conçu en prenant en compte la sécurité et qu'il soit utilisé correctement.

Dans ce chapitre, nous allons d'abord présenter le fonctionnement d'un compilateur usuel, quelques transformations d'optimisation classiques et le principe de fonctionnement d'un gestionnaire de passes. Puis, nous présenterons les limitations des compilateurs usuels pour la sécurité et en particulier pour l'application de transformations d'obfuscation. Enfin, nous présenterons quelques concepts de solutions possibles à ces problématiques de compilation utilisant des transformations de sécurité.

1. Particulièrement, s'ils sont utilisés avec les options par défaut ou usuelles.

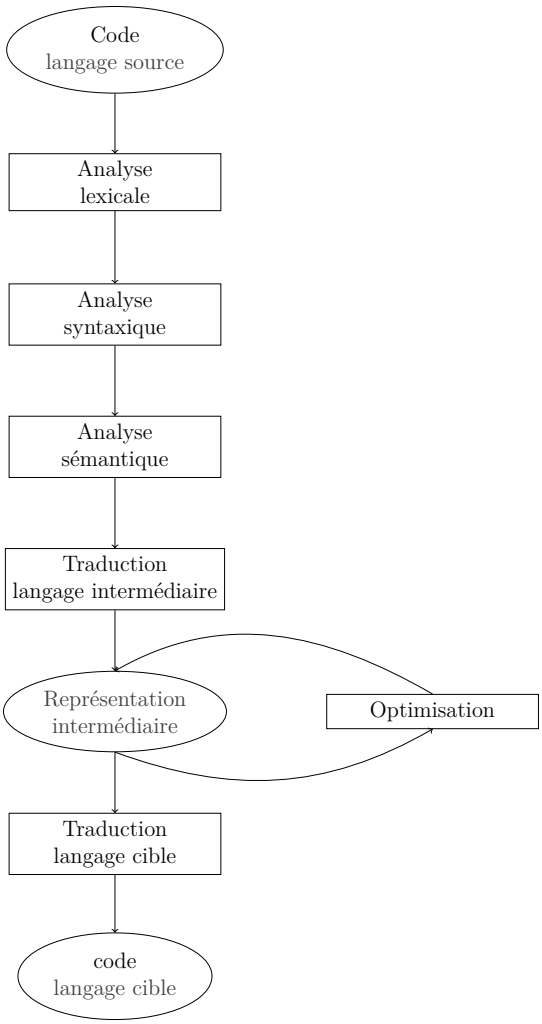

FIGURE 3.1 – Schéma de fonctionnement d'un compilateur

3.2 Processus traditionnel de compilation

3.2.1 Étapes de la compilation

Dans cette section, nous allons présenter brièvement les étapes qui sont généralement présentes dans le processus de compilation dans un compilateur traditionnel [2] pour voir comment l'obfuscation peut s'intégrer à ce processus. Ces différentes étapes sont représentées dans la figure 3.1 et décrites plus en détail ci-dessous.

Analyse lexicale

Un programme source est représenté par une suite de caractères, le code source. La première étape de la compilation, l'analyse lexicale, consiste à décomposer cette suite de caractères en séquences ayant un sens appelés *lexèmes*.

Les lexèmes sont convertis en *jeton* (*token*) pour l'analyse syntaxique. Les tokens sont de la forme : $\langle token\_name, attribute\_value \rangle$ où *token_name* est le symbole abstrait et *attribute_value* la valeur associée à ce symbole. La valeur n'est pas obligatoire, dans certains cas le symbole abstrait contient toute les informations. Par exemple, un identificateur (d'une variable, d'une fonction ...) aura un jeton de la forme $\langle id, 3 \rangle$, ce qui signifie qu'il s'agit d'un identifiant et plus précisément le numéro 3 de la table des identifiants (qui contient des informations sur la variable, comme son nom ou son type).

Analyse syntaxique

L'analyse syntaxique est la seconde étape de la compilation et elle consiste à transformer la suite de jetons créés à l'étape précédente en une représentation de la structure grammaticale du programme. On utilise en général un arbre de syntaxe abstraite comme représentation. Il s'agit d'un arbre dont les nœuds intérieurs sont les opérateurs et les enfants de ces nœuds sont les arguments de l'opérateur.

Analyse sémantique

Lors de l'analyse sémantique, le compilateur utilise les informations calculées précédemment pour vérifier que la sémantique du programme respecte la définition du langage. Par exemple, le compilateur vérifie que les variables sont correctement déclarées, que leur portée est respectée ou encore que les fonctions sont appelées correctement. L'analyse sémantique résout également les surcharges de fonction (*overload resolution.*) Enfin, l'analyse sémantique comprend souvent une vérification de type, c'est-à-dire que les arguments d'un opérateur sont du type qui correspond à la définition de l'opérateur. Par exemple, les arguments des opérateurs arithmétiques doivent être des nombres entiers ou flottants.

L'analyse sémantique peut également ajouter des informations supplémentaires à l'arbre de syntaxe. Par exemple, dans certains langages il peut y avoir des conversions automatiques de types pour certains opérateurs (par exemple, si on additionne un entier avec un flottant). Dans ce cas, le compilateur ajoutera un nœud de conversion à l'arbre de syntaxe

du programme.

Génération de code dans une représentation intermédiaire

Les compilateurs vont en général utiliser une ou plusieurs représentations intermédiaires au cours de la traduction entre le langage source et le langage cible. Il y a plusieurs intérêts à ces langages intermédiaires :

- Ils permettent de réutiliser les mêmes processus de compilation pour plusieurs langages sources et/ou plusieurs langages cibles. En effet, il suffit d'avoir une traduction des langages sources vers le langage intermédiaire et une traduction du langage intermédiaire vers chaque langage cible, plutôt que la traduction pour chacun des couples (langage source, langage cible) possibles.
- Ils permettent d'écrire des transformations de code (notamment d'optimisation) sur un langage ayant de bonnes propriétés pour l'écriture de ces transformations.
- Ils sont généralement plus faciles à traduire vers et depuis un autre langage.
- Ils peuvent permettre de faire certaines vérifications sur la correction de la compilation et des transformations. Par exemple, CompCert [140] utilise 8 langages intermédiaires pour assurer la correction de la compilation.

Optimisation

Cette étape de la compilation a pour objectif de rendre le programme final plus performant. Pour cela le compilateur applique des transformations de code sur la représentation intermédiaire pour rendre le programme plus performant. Il existe plusieurs critères de performances :

- La vitesse d'exécution du programme.
- La taille de l'exécutable du programme.
- L'espace mémoire utilisé par le programme.

Plusieurs transformations d'optimisation sont présentées en 3.2.2.

Génération du code cible

La dernière étape de la compilation est la traduction du langage intermédiaire vers le langage cible. Si le langage cible est un langage machine, il sera nécessaire à cette étape de choisir comment gérer la mémoire et gérer les emplacements de mémoire et de registres utilisés.

3.2.2 Transformations d'optimisation

Il existe de très nombreuses transformations d'optimisation avec des objectifs et des intérêts différents. Nous allons présenter ici quelques-unes des transformations les plus courantes dans les compilateurs.

Élimination du code mort

Dans un programme, il peut y avoir du code mort, c'est-à-dire des instructions qui ne sont soit jamais exécutées, soit qui font des calculs, dont les résultats ne sont jamais utilisés. Une des optimisations les plus simples est l'élimination du code mort (*Dead Code Elimination*) [2]. Le principe d'une version simple de cette optimisation est de créer le DAG (*Directed Acyclic Graph*) d'un *basic block* pour observer comment les variables du bloc sont calculées et utilisées. On peut supprimer toutes les variables qui ne sont pas utilisées dans le programme. Il existe des méthodes plus agressives [123][107][39] pour retirer le code mort d'un programme que nous ne détaillerons pas ici, car nous nous intéressons surtout au principe de cette optimisation.

Optimisation « Peephole »

Le principe des optimisations *peephole* est d'optimiser, non pas le programme dans sa globalité, mais localement en observant une séquence courte d'instructions (le *peephole*) et en changeant cette séquence en une séquence plus rapide ou plus courte [147][192]. Ce processus est itératif, on va déplacer la fenêtre, changer sa taille et passer plusieurs fois sur la même section (car faire une optimisation peut ouvrir la possibilité pour d'autres optimisations). Plusieurs transformations sont possibles dans les optimisations de *peephole* [55], nous allons en présenter quelques-unes.

Élimination des *loads* et *stores* redondants Une génération naïve de code (ou une autre transformation passée auparavant) peut générer des instructions de *load* (chargement d'une valeur de la mémoire vers un registre) et *store* (enregistrement d'une valeur d'un registre vers la mémoire) inutiles. En effet, on va souvent trouver dans un code, une séquence d'instruction de la forme :

```
LD addr, R0
ST R0, addr
```

L'enchaînement de ces deux instructions est inutile, car il charge une donnée de la mémoire dans un registre et sauvegarde cette même valeur dans le même emplacement mémoire. On peut donc supprimer ces deux instructions et gagner en performance [192]. Le cas présenté ici est trivial, mais cette optimisation est également applicable à des cas plus complexes [78][60]

Optimisation du flot de contrôle La génération de code va souvent générer des sauts qui mènent vers d'autres sauts. Ces sauts peuvent être des sauts directs ou des sauts conditionnels. Il est possible de supprimer certains de ces sauts en sautant directement à la destination [128]. Par exemple, le code suivant

```
        goto Label1
        /*...*/
Label1: goto Label2
```

peut être remplacé par

```
        goto Label2
        /*...*/
Label1: goto Label2
```

Simplification arithmétique Il est possible de supprimer certaines instructions arithmétiques inutiles [147]. Par exemple $x = x * 1$ est une instruction inutile qui peut être retirée du code du programme.

On peut également améliorer les performances d'un programme en remplaçant des calculs coûteux par des calculs équivalents utilisant des instructions moins coûteuses. Par exemple, une division par deux peut être remplacée par un décalage binaire (*shift*) et le calcul de x^2 sera en général beaucoup plus rapide en calculant $x * x$ plutôt que de faire l'exponentiation de x.

3.2.3 Gestionnaire de passes

Une passe de compilation est une transformation de code effectuée durant la compilation [2]. Une passe prend en entrée un programme (ou une partie d'un programme) et donne en sortie une nouvelle version de ce programme. Dans le cas général, une passe peut décrire n'importe quelle étape de la compilation, mais nous allons nous intéresser ici principalement aux passes s'exécutant sur la représentation intermédiaire du programme.

En effet, ces passes sont celles qui effectuent les transformations d'optimisation et les analyses sur le programme, et donc également une étape dans laquelle on peut insérer des transformations d'obfuscation supplémentaires.

Il existe de nombreuses transformations (ou passes) d'optimisation, et leur ordonnancement n'est pas trivial : les passes peuvent être appliquées plusieurs fois et l'ordre de l'application aura un impact sur les performances du programme. Certaines passes d'optimisation ont parfois besoin de connaitre certaines informations supplémentaires sur le programme : analyses d'aliasing, analyses de dépendances ... Ces analyses peuvent constituer des passes d'analyse.

Pour gérer l'ordonnancement des passes d'optimisation, les compilateurs contiennent en général un gestionnaire de passes [163][162][167]. Le gestionnaire de passes a plusieurs objectifs :

- Exécuter les passes dans le bon ordre. Cet ordre est souvent fixé et choisi par un expert.
- Gérer les passes d'analyse. On peut, par exemple, n'exécuter les analyses que lorsqu'une passe d'optimisation en a besoin.
- Gérer les options de compilation. La plupart des compilateurs proposent des options de compilation qui affectent le processus d'optimisation : `-O1`, `-O2` ...

Les compilateurs principaux possèdent des gestionnaires de passes avec une structure relativement rigide et assez peu dynamique dans l'exécution du processus d'optimisation. Cela est en général suffisant pour réaliser l'optimisation du programme, mais peut être plus limitant dans le domaine de la protection de logiciel comme va le montrer la suite de ce chapitre.

3.3 Limitations des compilateurs pour la protection de logiciel

3.3.1 Introduction

En général, les compilateurs sont conçus sans prendre en compte les problématiques de protection de logiciel. En effet, leurs objectifs sont de compiler le programme en conservant sa sémantique, d'être efficaces dans la compilation (performance de la compilation en vitesse et mémoire) et de générer un programme cible performant (en vitesse, taille et/ou mémoire). Dans cette partie, nous allons voir que la protection de logiciel impose des

problématiques supplémentaires pour lesquelles ces compilateurs sont peu adaptés.

3.3.2 Interactions optimisations-obfuscations

On cherche en général à obtenir les meilleures performances possibles pour un programme, même si l'on veut le protéger avec de l'obfuscation ou d'autres techniques de protection de logiciels. Ce qui signifie en général que l'on va appliquer les transformations classiques d'optimisation pendant le processus de compilation et d'obfuscation.

On peut appliquer les transformations d'optimisation avant les transformations d'obfuscation, après les transformations d'obfuscation ou entrelacer ces transformations. Ces choix auront un impact à la fois sur les performances du programme et sur son niveau de protection [105].

Dans le cas où l'on choisit d'appliquer les transformations d'optimisation avant celles d'obfuscation, les passes d'optimisation pourront généralement s'appliquer à leur potentiel maximum. En effet, certaines transformations d'obfuscation peuvent empêcher les optimisations de s'appliquer efficacement. Par exemple, la virtualisation va transformer le programme en une donnée d'une machine virtuelle [20][67]. Les optimisations usuelles travaillant sur le code du programme et non pas sur les données, on pourra donc optimiser le code de la machine virtuelle, mais pas celui du programme qu'elle exécute.

De plus, pour pouvoir s'appliquer certaines optimisations ont besoin de « comprendre » le programme, et pour cela utilisent certaines analyses sur le code du programme. Or le but de certaines obfuscations est d'empêcher des outils automatiques de résoudre certains problèmes (par exemple analyse d'alias [144] ou prédicats opaques [73]), pour empêcher la désobfuscation automatique.

De la même façon, les optimisations peuvent avoir un effet sur la sécurité apportée par les obfuscations en simplifiant certaines parties du code, qui avaient été complexifiées volontairement pour améliorer la sécurité. Mais, en général, on considère que si la protection apportée par une obfuscation est retirée par une optimisation classique, il ne s'agit pas d'une bonne protection. En effet, cela signifie qu'un attaquant pourra la retirer relativement simplement avec un outil automatique. Cela signifie que cette protection ne doit pas être utilisée, ou être utilisée en conjonction d'autres protections qui empêchent les ces outils automatiques de fonctionner. De plus, l'efficacité de certaines transformations d'obfuscation est fortement dépendante de la structure du programme (cf. 2.5.1, 2.5.2), or les optimisations peuvent changer la structure du programme et donc modifier l'efficacité de la protection apportée par l'obfuscation.

En général, le code généré par les transformations d'obfuscation est écrit pour être sécurisé et non pas pour être rapide. Ce qui veut dire que ce code est très souvent optimisable. En pratique, il sera souvent très intéressant d'optimiser après l'ajout de la protection pour optimiser le code modifié et généré par les transformations d'obfuscation. Comme détaillé précédemment, il sera aussi parfois intéressant d'optimiser le programme avant l'obfuscation. Mais le processus optimal pour le compromis performance/sécurité sera souvent de mêler les transformations d'optimisation et d'obfuscation pour limiter les interactions négatives et augmenter les interactions positives entre les transformations. Ce processus sera bien sûr plus coûteux en temps de développement (pour exprimer la chaîne de compilation voulue), et ne sera donc pas adapté à tous les usages.

3.3.3 Incompatibilités de transformations

Certaines transformations de code ne peuvent pas être appliquées simultanément sur un même programme, soit parce que cela rendrait le code du programme invalide, soit parce que l'une (ou plusieurs) de ces transformations n'apportera pas de modifications sur le code.

Les problèmes de génération de code invalide surviennent souvent avec les transformations de *tamperproofing*, car ces transformations changent la sémantique du programme. Elles modifient la sémantique afin d'apporter une réponse à une attaque (p. ex. *crash* le programme). Comme cela a été expliqué en 2.3.3, les fonctions de détection d'attaques se basent en général sur l'état du programme et sa mémoire. Or, les transformations d'obfuscation peuvent modifier ces données, et donc appliquer ces transformations d'obfuscation après l'application des protections de *tamperproofing* peut créer de faux positifs (pour la détection d'attaques). Les faux positifs provoqueront un déclenchement de la réponse dans des cas d'utilisation normale du programme, ce qui n'est pas voulu. La génération de code invalide peut aussi arriver avec seulement des transformations d'obfuscation, car certaines d'entre elles supposent aussi que le programme est dans un certain état (celui où il était au moment de l'application de la transformation). Par exemple, les obfuscations utilisant du *Return-Oriented Programming* (ROP) [174][160][143], c'est-à-dire notamment qu'elles changent dynamiquement l'adresse de retour des fonctions pendant leurs exécutions pour créer un chemin d'exécution difficile à déterminer statiquement, ont besoin de connaitre les adresses de certaines sections du code [2]. Si une autre transformation de code

2. Ces transformations sont généralement directement appliquées sur le code exécutable pour cette raison.

est appliquée naïvement par la suite, il y a de grandes chances que ces adresses deviennent incohérentes.

3.3.4 Impact de l'ordre d'application des transformations sur le niveau d'obfuscation

L'ordre des transformations peut changer le niveau de protection d'un programme, comme elle affecte également fortement les performances d'un programme lors de l'application de transformations d'optimisation, ce qui a été beaucoup étudié pour la conception des chaînes d'optimisation des compilateurs [89][30][10][94].

Par exemple, pour une transformation classique d'obfuscation telle que la substitution des opérateurs binaires standards par des séquences d'instructions plus compliquées, mais fonctionnellement équivalentes l'application d'optimisations après cette passe retire en général la substitution, car il s'agit d'une protection qui est assez peu résistante contre les attaques automatiques. La figure 3.2.2 montre un exemple de l'interaction entre les optimisations, la substitution et les constantes opaques. L'optimisation retire la protection apportée par la substitution si elle est appliquée seule. Par contre, si elle est combinée avec une transformation de constantes opaques, alors l'optimisation ne retire plus la protection apportée par la substitution. En effet, la passe de constantes opaques transforme les constantes pour qu'elles ne soient pas identifiables ou calculables facilement. Le niveau de sécurité apportée par la suite de transformations : « Substitution - Constantes opaques - Optimisations » est donc plus élevé que pour la séquence : « Substitution - Optimisations - Constantes opaques ».

L'exemple présenté ici n'est pas un cas particulier et ce type d'interactions entre les passes d'obfuscation est très courant [161] et possède un effet important sur le niveau de protection du programme [142][113].

3.3.5 Limitations des compilateurs dans la protection contre les attaques matérielles

Les programmes peuvent également être victimes d'attaques matérielles, telles que les attaques par injection de fautes [40] qui introduisent des comportements non désirés dans le programme en insérant des disruptions matérielles. Par exemple, en changeant la tension, la température, la vitesse de l'horloge ou en utilisant des lasers [24]. Une des attaques possibles est l'*instruction skip*, qui consiste à forcer le matériel à passer une

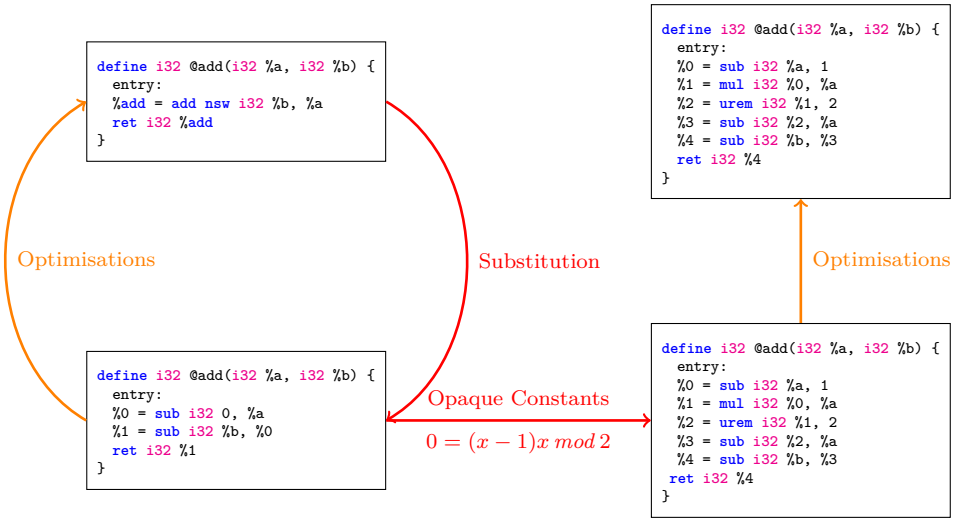

FIGURE 3.2 – Impact de l'ordre des transformations sur la protection

instruction du programme sans l'exécuter grâce à une disruption matérielle [43][180]. Cela modifie le comportement du programme et peut permettre, par exemple, à un attaquant de s'authentifier sur un système avec PIN avec un faux PIN [87] ou d'obtenir des informations sur une clé de chiffrement [194].

Il est possible de se protéger de ce type d'attaques matérielles grâce à des protections logicielles (en plus des protections matérielles [103][181]). Il est possible d'intégrer ces protections au niveau du code source, pour détecter ou corriger les fautes [127]. Comme les autres transformations de *tamperproofing*, elles dépendent du code au moment de l'insertion des protections et sont donc sensibles aux autres transformations appliquées (cf. 3.3.3). Une solution intuitive contre les attaques de saut d'instructions est de dupliquer les instructions [28][157] dans les cas où cela ne change pas la sémantique du programme. Et comme cela ne change pas la sémantique du programme, un compilateur retirera en général l'instruction en double et retirera donc la protection.

Comme pour l'obfuscation, les compilateurs traditionnels ne sont pas forcément adaptés à l'application de protection contre les attaques matérielles. Il peut donc être intéressant de modifier un compilateur et son gestionnaire de passes pour prendre en compte la protection contre les attaques matérielles [31]. Les contributions de cette thèse sont centrées autour de l'obfuscation, mais elles pourraient également s'appliquer en grande partie à la protection contre les attaques matérielles.

3.4 Réponses à ces problématiques

Dans la partie précédente, nous avons présenté les limitations des compilateurs traditionnels pour la protection de logiciel. Cette partie va présenter quelles sont les possibilités pour la compilation de programmes avec de la protection de logiciel.

3.4.1 Obfuscation manuelle

La première solution est d'effectuer la compilation et l'obfuscation en grande partie manuellement par un expert en obfuscation. L'expert pourra choisir quelles les transformations seront appliquées, dans quel ordre et où elles sont appliquées. Il choisira également les paramètres adaptés au programme.

Grâce à son expertise, il pourra éviter les incompatibilités et les interactions destructives entre les transformations d'obfuscation et d'optimisation. Il pourra également adapter le processus de compilation aux programmes cibles afin d'obtenir le meilleur compromis entre la sécurité et les performances.

Cette méthode est par contre très coûteuse, et des outils permettant d'automatiser une partie du processus, donnant des informations supplémentaires ou effectuant certaines vérifications seront aussi utiles à un expert et permettront d'éviter certaines erreurs et d'obtenir une obfuscation de meilleure qualité.

3.4.2 Obfuscateur automatique

Un obfuscateur automatique est un outil permettant d'obfusquer un programme automatiquement. Il s'agit d'un programme qui prend en entrée le programme à protéger et (parfois) des paramètres supplémentaires de configuration. La sortie de l'obfuscateur automatique sera le programme protégé.

L'intérêt de ces obfuscateurs automatiques est qu'ils permettent à des utilisateurs, qui ne sont pas experts en obfuscation, de protéger leurs programmes. Cela offrira par contre un processus d'obfuscation plus générique qu'une obfuscation manuelle, et donc probablement une moins bonne atteinte des objectifs de sécurité et de performance. Cependant, ces outils ont été développés par des experts en obfuscation et ont été testés sur un grand nombre de programmes et produisent donc des chaînes de compilation qui ne devraient pas créer de problèmes de comptabilité et d'interactions entre les transformations. Ces obfuscateurs contiennent également souvent des outils qui permettent aux experts de guider

l'obfuscation du programme, par exemple, à l'aide de *pragma* inséré dans le code source.

Il y a plusieurs obfuscateurs automatiques utilisés industriellement. Nous allons présenter les fonctionnalités de deux d'entre eux. *Obfuscator-llvm* [119] est un obfuscateur automatique qui se base sur LLVM et s'exécute sur le langage intermédiaire de LLVM. Cela lui permet de fonctionner sur de nombreux langages (C, C++, Fortran, Julia, Swift et tous les autres langages qui possèdent un *front end* LLVM) et sur plusieurs plateformes matérielles (ARM, x86, SPARC). Il met à disposition plusieurs protections à l'aide de drapeau à ajouter à la commande de compilation notamment : substitution d'instructions, *control flow flattening* (cf. 2.5.2), *bogus control flow* (cf. 2.5.1) et des techniques de *tamperproofing*.

Tigress [65] est un obfuscateurs source à source, c'est-à-dire qu'il prend en entrée un code source en C et renvoie un code source C obfusqué. Tigress comprend de très nombreuses transformations sur le code comme la virtualisation (cf. 2.5.4), les prédicats opaques ou le *control flow flattening*. Tigress permet d'utiliser de très nombreuses options dans le choix des transformations et leurs paramètres d'application grâce à des drapeaux de compilation. En particulier, Tigress permet de cibler les fonctions à transformer et de choisir l'ordre des transformations, et également de fixer les graines des générateurs aléatoires (pour la reproductibilité du processus). Cela fait qu'il est nécessaire d'écrire un script d'obfuscation et donc d'avoir de bonnes connaissances dans le domaine de l'obfuscation et dans la syntaxe et la sémantique de Tigress.

Il existe d'autres solutions commerciales telles que VMProtect [196] qui utilise la virtualisation pour protéger des binaires, PELock qui protège des binaires contre les attaques de type *tampering*, ASPack Software propose ASProtect [12] qui protège les binaires avec des techniques de cryptographie et ASObfuscator [11] qui est un obfuscateur source à source pour les programmes C/C++. Enfin Epona [90] est un obfuscateur propriétaire travaillant sur la représentation intermédiaire LLVM.

Il est possible d'utiliser plusieurs de ces outils sur un même programme pour bénéficier des différentes fonctionnalités. Cependant, ces outils sont pensés comme des solutions complètes, donc des expérimentations ou une expertise peuvent être requises pour obtenir un résultat intéressant. En particulier, la composition naïve de ces outils peut dégrader les performances d'un programme à un niveau inacceptable sans augmentation notable de la protection (cf. 2.3.7).

3.4.3 Obfuscation executive

Un *Obfuscation Executive* [111] est un outil d'obfuscation qui applique un ensemble de transformations automatiquement en boucle jusqu'à atteindre un certain objectif. L'*obfuscation executive* demandera peu d'entrées à l'utilisateur, c'est-à-dire, en général :

— Quels sont les objectifs de performances et de protections ?
— Quelles sont les parties du code à protéger ?
— Quelles sont les parties du code critique pour les performances ?

Comme un optimiseur de code, il permet de résoudre, de façon automatique, deux problématiques de l'obfuscation :

— L'ordonnancement des transformations. Pour un optimiseur, l'ordre est en général fixé dans le compilateur par des experts.
— La condition d'arrêt de la boucle de transformations. Dans un optimiseur, c'est simplement une fois que l'ensemble des transformations prévues ont étés appliquées.

Mais comme nous l'avons vu en 3.3, dans le cas de l'obfuscation, on peut difficilement proposé un script d'obfuscation "fixe" contrairement à l'optimisation. Le rôle de l'*obfuscation executive* est donc très important pour résoudre ces problématiques automatiquement.

Pour résoudre le problème d'ordonnancement, l'*obfuscation executive* prend en compte certaines informations sur les transformations : préconditions, postconditions, présuggestions et postsuggestions, qui sont vues plus en détail en 5.3, et des analyses sur le programme sur lequel les transformations sont exécutées. Pour la condition d'arrêt, il est nécessaire d'avoir une mesure de la qualité de l'obfuscation et de l'impact sur les performances, ce qui est difficile à obtenir (cf. 2.7), et on utilisera donc en général des valeurs qualitatives.

Ahmadvand et al. [1] propose un système similaire pour déterminer la composition de transformations à appliquer pour optimiser le compromis performance-sécurité avec une orientation vers le *tamperproofing*. Leur solution utilise un graphe (*defence graph*) pour représenter les contraintes et les conflits dans l'application des passes. Il considère également plusieurs catégories de transformations (par exemple, les transformations de hachage, celles d'autovérification, d'optimisation ...) et tente de respecter un ordre (fixé) dans l'application de ces types de passes tout en respectant les contraintes. Ils convertissent ensuite les contraintes en contraintes entières et utilisent la programmation linéaire en nombres entiers (ou *Integer Linear Programming* (ILP)) [183] pour résoudre ces contraintes.

Idéalement, l'*obfuscation executive* exécutera la chaîne de transformations optimale,

mais cela est peu probable à cause de la difficulté de mesurer l'impact des transformations et leurs ordres d'applications sur la protection et les performances [113]. Le résultat sera donc généralement moins bon que si l'obfuscation est réalisée par un expert, mais beaucoup moins coûteux (sur le plan du temps de développement de la chaîne d'obfuscation) [111].

3.4.4 Langage dédié

Principe

Pour exécuter le processus de compilation et d'obfuscation, on peut également écrire un script d'obfuscation. Celui-ci décrira l'ensemble des transformations à appliquer, comment elles doivent être appliquées et avec quels paramètres.

Le script d'obfuscation peut être écrit de plusieurs façons : il peut s'agir d'un script écrit dans un langage de script (Python, Shell ...) qui appelle une suite de ligne de commande pour piloter d'autres outils d'obfuscation et de compilation. Il peut également être intégré directement dans le compilateur, par exemple dans son gestionnaire de passes. Cela permet d'automatiser une partie du travail (décrit en 3.4.1) d'un expert et, avec des bibliothèques ou des fonctions aides, de faciliter et d'accélérer l'écriture du script.

On peut pousser ce concept et développer un langage spécifiquement dédié à l'écriture de programme d'obfuscation et proposant de nombreuses fonctionnalités facilitant l'écriture de scripts de compilation et vérifiant la correction des scripts.

Langages dédiés à l'optimisation et à la compilation

Il existe plusieurs langages de ce type dédiés à l'optimisation et la compilation dans certains domaines. Ces langages ne sont pas pensés pour l'obfuscation de logiciel, mais certains de leurs concepts peuvent être réutilisés dans un langage dédié à l'obfuscation.

Halide : Halide [171] est un langage utilisé pour le traitement d'images. Le processus de traitement d'images peut contenir de nombreuses boucles qui sont difficiles à optimiser avec les méthodes traditionnelles. Halide sépare le processus en un algorithme et un ordonnancement, c'est-à-dire quels sont les calculs et comment (dans quel ordre) ces calculs sont effectués. Les fonctions dans Halide calculent la valeur d'un pixel à partir de ses coordonnées sans effets de bords. Ces fonctions sont composées pour générer le programme Halide, qui n'a donc pas à gérer la gestion de la mémoire du programme final.

Le compilateur s'occupera de la gestion de la mémoire et des optimisations à partir du processus de traitement et de l'ordonnancement décrit séparément.

PyPS : PyPS [104] est une interface programmable du gestionnaire de passes pour le compilateur PIPS [117]. PIPS est un compilateur source à source pour la parallélisation de code. Il propose des abstractions de haut niveau pour la compilation (passes, fonctions, boucles ...) qui peuvent être utilisées dans un script Python pour décrire le processus de compilation avec plus de précision et de modularité. Il permet aussi d'utiliser des constructions logiques supplémentaires pour décrire le processus de compilation, telles que des boucles et des structures conditionnelles.

CHiLL : Pour obtenir le processus d'optimisation de boucle le plus performant, CHiLL [57] utilise un ensemble de transformations d'optimisation, les combine et choisit leurs paramètres pour créer une optimisation empirique performante. Un algorithme de décision crée un ensemble de scripts d'optimisation avec des paramètres non fixés puis un algorithme de recherche choisit les paramètres et mesure les performances de ces scripts pour trouver ceux qui obtiennent les meilleures performances.

3.5 Conclusion

Dans ce chapitre, nous avons présenté le fonctionnement des compilateurs usuels et des transformations d'optimisation et nous avons montré que ces compilateurs possèdent des limitations et sont généralement peu adaptés à l'utilisation de transformation d'obfuscation. Il est donc nécessaire d'adapter les compilateurs pour produire des programmes sécurisés. Pour cela, nous avons présenté plusieurs possibilités pour produire du code obfusqué.

Pour diverses raisons, qui seront décrites en détail dans les chapitres 5 et 6, nous avons décidé de nous concentrer sur la création d'un langage dédié à la conception de chaînes d'obfuscation logicielle.

Chapitre 4

Langages fonctionnels et leur sémantique

4.1 Introduction

Maintenant que nous avons présenté le contexte autour des travaux de la thèse, particulièrement les problématiques de la conception de chaînes de compilation contenant des transformations d'obfuscation, nous allons présenter certains outils existants sur lesquels nous allons nous baser pour répondre à ces problématiques.

Comme nous l'avons indiqué précédemment, la contribution principale de cette thèse est SCOL, un langage dédié à la conception de chaînes de compilation. Ce chapitre a pour objectif de présenter certains travaux existant pour la création de langage de programmation que nous utiliserons dans SCOL.

En particulier, nous allons présenter le principe de fonctionnement et les règles syntaxiques et sémantiques du λ-calcul. Dans la suite de ce chapitre, nous présenterons également différents systèmes de types. Nous décrirons l'utilité de ces systèmes de types et leurs principes de fonctionnement. Nous présenterons également les règles de typage usuelles d'un λ-calcul avec ces différents systèmes de types. Dans ce chapitre, nous ne faisons pas une présentation exhaustive de toutes les possibilités pour la conception de langage, mais nous présentons avec plus de détails les concepts qui nous seront utiles pour réaliser notre langage dédié.

4.2 Un langage fonctionnel

4.2.1 Intérêts

Pour raisonner sur les propriétés syntaxiques et sémantiques des programmes, il est nécessaire de pouvoir faire des opérations vérifiables mathématiquement sur les programmes.

Les langages fonctionnels possèdent des sémantiques formelles simples et facilitent la réalisation de preuves formelles sur la compilation et l'exécution des programmes [152]. De plus, les langages fonctionnels limitent les effets de bords dans les programmes, qui peuvent être complexes à analyser (le comportement dépend du contexte), il est donc plus simple de vérifier formellement le comportement du programme et de créer des programmes réutilisables.

4.2.2 λ-calcul

Le λ-calcul [169][27] est un système formel, proposé par Alonzo Church dans les années 1920 [63], qui réduit tous les calculs à des opérations de définitions et d'applications de fonctions. Il s'agit à la fois d'un objet mathématique sur lequel des propriétés peuvent être prouvées et d'un langage qui peut décrire les calculs.

De plus, ce langage est extensible, des fonctionnalités peuvent être ajoutées, ce qui permet de créer un langage adapté à notre besoin.

Le λ-calcul comprend uniquement 3 termes simples : une variable x, l'abstraction du terme e par la variable x noté $\lambda x.e$ et l'application d'un terme e_1 à un autre terme e_2 noté $e_1\ e_2$. La grammaire de ces termes peut donc se résumer ainsi :

$$e ::= x \mid \lambda x.e \mid e\ e$$

Dans l'abstraction $\lambda x.e$ la variable x est liée pour ses occurrences dans le corps e.

Ici, notre seule opération de calcul sera l'évaluation par valeur de l'application d'une fonction sur un argument. Cela est réalisé en substituant la variable liée (x) par l'abstraction (terme de gauche) dans son corps (e) par la valeur de l'argument de l'application (terme de droite). La notation que nous utiliserons est la suivante :

$$\textsc{App}\ \frac{e_2 \Downarrow v_1 \qquad [x \mapsto v_1]e_1 \Downarrow v_2}{\lambda x.e_1\ e_2 \Downarrow v_2}$$

Cette règle est appelée l'application [1]. Cette notation signifie que si l'expression e_2 s'évalue en la valeur v_1 et l'expression e_1 s'évalue en la valeur v_2 en remplaçant la variable x par la valeur v_1 dans l'expression e_1 (noté $[x \mapsto v_1]e_3$) alors l'expression $\lambda x.e_1\ e_2$ s'évalue

1. Avec cette règle nous présentons ici un langage moins général que le λ-calcul, car nous utilisons un appel par valeur plutôt que la β-reduction. Nous le présentons comme cela, car c'est ce que nous utiliserons ensuite.

en la valeur v_2.

Les valeurs sont les résultats possibles d'une évaluation d'un programme. Dans le λ-calcul, dans sa version très simple présentée ici, les seules valeurs possibles sont les abstractions :

$$v ::= \lambda x.e$$

Nous intégrons aussi la construction `let` qui est une règle dérivée de la combinaison de l'abstraction et de l'application. On peut réécrire $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ sous la forme $(\lambda x.e_2)\ e_1$. Informellement, cela signifie que l'on associe le terme e_1 à la variable x dans le terme e_2. La sémantique de cette construction est la suivante :

$$\textsc{Let}\ \frac{[x \mapsto e_1]e_2 \Downarrow v_2}{\texttt{let}\, x = e_1\, \texttt{in}\, e_2 \Downarrow v_2}$$

4.2.3 Exemples

Bien que la syntaxe de ce langage est minimale, on peut tout de même écrire des programmes dans ce langage. Notons que les variables libres sont des termes valides du langage, cela permet, par exemple, de définir des valeurs constantes dans le langage. Par exemple, pour décrire les entiers naturels on utilise souvent les nombres de Church [62], qui s'écrivent de la façon suivante :

$$\begin{aligned} 0 &:= \lambda f.\lambda x.x \\ 1 &:= \lambda f.\lambda x.f\ x \\ 2 &:= \lambda f.\lambda x.f\ (f\ x) \\ 3 &:= \lambda f.\lambda x.f(f\ (f\ x)) \end{aligned}$$

Et ainsi de suite pour les nombres suivants.

Cette notation permet d'écrire la fonction calculant le successeur d'un nombre entier naturel. En effet, la fonction $\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$ appliquée à un nombre entier n donne la valeur $n + 1$ dans la notation de Church.

De la même façon, on peut définir la fonction $\lambda m.\lambda n.\lambda f.\lambda x m\ f(n\ f\ x)$ calcule la somme $m + n$ lorsqu'elle est appliquée à deux arguments entiers naturels m et n.

On pourrait bien sûr décrire les autres opérations arithmétiques de cette façon, ou

encore décrire d'autres types de calculs comme les opérations logiques [2], mais nous ne les utiliserons pas par la suite.

4.3 Systèmes de types

4.3.1 Intérêts et propriétés

Définitions

Le respect de la syntaxe d'un langage fonctionnel n'assure pas qu'il n'y aura pas d'erreurs d'exécution et ne garantit pas que le calcul s'exécute jusqu'à l'obtention d'un résultat (possibilités de blocages ou de divergences du calcul). Il est utile de savoir si l'exécution permettra d'obtenir un résultat avant l'évaluation du terme (donc statiquement).

Un système de types permet d'atteindre ces objectifs [169]. Le terme t a pour type τ (noté $t : \tau$) s'il peut être évalué statiquement que le terme t a une forme correspondante au type τ, par exemple t est un entier ou t est un booléen. Soit l'assertion $\Gamma \vdash t : \tau$, qui signifie que le terme t a pour type τ dans le contexte de typage Γ. Le contexte de typage est une fonction associant un type à une variable. Lors de l'évaluation du type d'un terme, lorsqu'une information de type d'une variable sera calculée (en particulier, par une abstraction), elle sera ajoutée au contexte de typage pour être utilisée dans l'évaluation du type d'autres termes.

Un système de types est un ensemble de règles de typage qui déterminent le type d'un terme en fonction de celui des termes le composant. Ces règles permettent d'évaluer le type d'un terme (inférence de type). Un langage est dit typé s'il possède un système de types.

Les programmes peuvent être typés grâce à deux techniques différentes : le typage statique et le typage dynamique.

Dans le cas du typage statique, le type est déterminé avant l'exécution du programme soit, car le type est explicitement spécifié par le programmeur, soit, car il peut être inféré à partir du code du programme. Dans tous les cas, les erreurs de types sont détectées avant l'exécution du programme.

Dans le cas du typage dynamique, le type est uniquement déterminé à l'exécution du programme, il ne peut donc pas y avoir de vérification statique des erreurs de types.

2. Le λ-calcul est Turing complet [195]

Le typage statique améliore la sûreté du programme, mais complexifie son développement. Il est parfois difficile de typer statiquement les variables : par exemple, certaines fonctions peuvent renvoyer soit un résultat, soit une erreur, qui n'ont pas le même type (au moins d'un point de vue sémantique).

Propriétés des langages typés

Définition. Un terme t est *typable* (ou *bien typé*) dans le contexte de typage Γ s'il existe un type τ tel que $\Gamma \vdash t : \tau$.

Définition. Un langage typé est dit *sûr* (*safe* ou *sound*) si les termes bien typés ne peuvent pas « mal s'exécuter » (« Well-typed terms do not "go wrong". » [151]), c'est-à-dire que le calcul du terme ne se retrouve pas dans un état bloqué.

La sûreté d'un langage typé se décompose, en général, en deux propriétés, la propriété de *progression* (un terme bien typé est soit une valeur soit peut exécuter un pas d'évaluation) et celle de *préservation* (le résultat d'un pas d'évaluation d'un terme bien typé est un terme bien typé et le type de ce terme est conservé).

Une autre propriété intéressante dans un langage est de savoir si l'évaluation d'un terme s'arrête en un nombre fini de pas d'évaluation (il n'y a pas de boucles), c'est-à-dire que tous les termes bien typés sont *normalisables*.

4.3.2 λ-calcul simplement typé

Nous pouvons maintenant définir le λ-calcul simplement typé dont la syntaxe et la sémantique sont détaillées ci-dessous : [169]

$$e ::= x \mid \lambda : \tau.e \mid e\ e \qquad v ::= \lambda x : \tau.e \qquad \tau ::= \sigma \mid \tau \to \tau$$

Comme nous pouvons le voir ici, la syntaxe est identique à celle du λ-calcul que nous avons présenté précédemment, à la différence que nous précisons le type de la variable x dans l'abstraction, avec la notation $\lambda : \tau.e$ qui indique que la variable x aura le type τ dans le terme e.

$$\textsc{App}\ \frac{e_1 \Downarrow \lambda x : \tau . e_3 \qquad e_2 \Downarrow v_1 \qquad [x \mapsto v_1]e_3 \Downarrow v_2}{e_1\ e_2 \Downarrow v_2}$$

De même, la règle App est identique (à la syntaxe de l'abstraction près) à la règle équivalente dans le λ-calcul non typé.

Le système de types du λ-calcul simplement typé est composé de trois règles qui sont les suivantes :

$$\textsc{TVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad\qquad \textsc{TLam}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

La règle TVar signifie que si une variable est présente dans le contexte de typage Γ, alors cette variable est du type qui lui est associé dans le contexte. La règle TLam indique que si lorsque x est de type τ_1 dans e, e est de type τ_2, alors l'abstraction de x dans e est une fonction de type $\tau_1 \to \tau_2$. Enfin, la règle TApp, indique que l'argument d'une application (e_2) doit être identique au type de l'entrée de l'abstraction τ_1. Dans ce cas, l'application peut être réalisée, et le résultat sera du type du résultat de l'abstraction τ_2.

4.3.3 Sous-typage

Les règles de typage peuvent être assouplies en utilisant le sous-typage [75][5]. Un type τ_1 est un sous-type du type τ_2 (noté $\tau_1 <: \tau_2$) si τ_1 peut être utilisé alors que τ_2 est attendu.

Ceci est défini formellement par la règle de *subsumption* :

$$\textsc{TSub}\ \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

Cette règle (TSub) signifie qu'il est possible de considérer un terme, comme étant d'un

sur type de son type réel. Autrement dit, on peut par exemple, appliquer une fonction sur un argument qui est un sous-type du type d'entrée de la fonction [3].

De plus le sous-typage est une relation réflexive (règle Ref) et transitive (règle Trans) :

$$\textsc{Ref}\ \tau <: \tau \qquad\qquad \textsc{Trans}\ \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Pour illustrer l'intérêt du sous-typage, nous pouvons prendre comme exemple l'addition de nombres complexes : il s'agit d'une fonction de type $\mathbb{C}^2 \to \mathbb{C}$, mais cette fonction peut également être appliquée sur des nombres réels ou des entiers naturels. Ce qui donne les règles de sous-typage :

$$\mathbb{R} <: \mathbb{C} \qquad\qquad \mathbb{N} <: \mathbb{C}$$

Nous définissons également le sous-typage pour le type fonction $\tau \to \tau$. Cette règle est la suivante :

$$\tau_1 \to \tau_2 <: \tau_3 \to \tau_4 \Leftrightarrow \tau_3 <: \tau_1 \wedge \tau_2 <: \tau_4$$

Les fonctions sont donc contravariantes sur le type d'entrée et covariantes sur le type de sortie. Cela permet au programme de rester bien typé lorsqu'une fonction est remplacée par une fonction sous-type de la première. Si on reprend l'exemple des nombres réels et complexes, alors le type $\mathbb{C} \to \mathbb{R}$ sera un sous-type de $\mathbb{R} \to \mathbb{C}$.

4.3.4 Typage utilisant des rangées

Introduction

Nous allons présenter maintenant un système de types qui utilise les rangées pour fournir une notion d'ensemble. Les rangées créent un type qui permet de stocker un nombre infini d'informations [4] dans un type. Une rangée contient un ensemble (finis ou non) d'atomes et leur associe une **présence**. Cette présence indique si l'atome est présent ou absent de la rangée. Une des spécificités des rangées est qu'il est possible d'indiquer que la présence d'un atome dans un type est inconnue.

Dans cette partie , nous allons détailler la construction, le fonctionnement et le système de types associés aux rangées.

3. En utilisant la règle TSub sur le terme e_2 de la règle TApp.
4. Si la rangée est de taille infinie.

Les enregistrements

Soit $\mathbb{L}$ un ensemble dénombrable de labels.

Nous commençons par définir les enregistrements (ou *records*) [44][164] sur $\mathbb{L}$. Il s'agit d'un ensemble (fini) d'éléments e_i de $\mathbb{L}$ possédant un type τ_{e_i}. On note cet enregistrement :

$$r ::= \{l_i = v_i\}_{i \in I}$$

Puis nous définissons une présence pour les enregistrements notés : $\{l : \pi\}_{l \in \mathbb{L}}$ où π est une des présences suivantes : Abs qui signifie que le label correspondant n'est pas présent dans l'enregistrement, ou $\mathsf{Pre}\ \tau$ qui signifie que l'élément est présent et de type τ.

Nous pouvons maintenant définir la syntaxe d'un λ-calcul utilisant des enregistrements :

$$e ::= x \mid v \mid e\ e \qquad v ::= \lambda x :\ \tau.e \mid r \qquad \tau ::= \sigma \mid \tau \to \tau \mid \{l : \pi\}_{l \in \mathbb{L}}$$
$$\pi ::= \mathsf{Abs} \mid \mathsf{Pre}\ \tau$$

Nous ne donnons pas ici les constructeurs et la sémantique pour les enregistrements, car nous nous intéressons uniquement au système de types.

La règle de typage des enregistrements est la suivante :

$$\textsc{TRec}\ \frac{L \subset \mathbb{L} \qquad \forall l \in L, \Gamma \vdash a_l :\ \tau_l}{\Gamma \vdash \{(l = a_l)_{l \in L}\} : \{(l : \mathsf{Pre}\,\tau_l)_{l \in L}; (l : \mathsf{Abs})_{l \notin L}\}}$$

Pour chaque label d'un enregistrement, on attribue $\mathsf{Pre}\,\tau$ où τ est le type correspondant à la valeur si ce label est défini dans l'enregistrement et Abs si ce label n'est pas défini dans l'enregistrement

Exemples : Pour expliciter le fonctionnement du typage avec des exemples, nous nous plaçons dans un cas simple avec un ensemble de labels de taille réduite $\mathbb{L} = \{\mathsf{e}, \mathsf{f}, \mathsf{g}\}$. Les types de plusieurs enregistrements sont donnés en exemples ci-dessous :

- Le type de l'enregistrement vide $\{\}$ est $\{\mathsf{e} : \mathsf{Abs}; \mathsf{f} : \mathsf{Abs}; \mathsf{g} : \mathsf{Abs}\}$.
- Le type de l'enregistrement $\{\mathsf{e} = 1; \mathsf{g} = \mathtt{true}\}$ est $\{\mathsf{e} : \mathsf{Pre}\,\mathtt{int}; \mathsf{f} : \mathsf{Abs}; \mathsf{g} : \mathsf{Pre}\,\mathtt{bool}\}$.
- Le type de l'enregistrement $\{\mathsf{e} = \mathtt{false}; \mathsf{f} = \mathtt{true}; \mathsf{g} = 0\}$ est $\{\mathsf{e} : \mathsf{Pre}\,\mathtt{bool}; \mathsf{f} : \mathsf{Pre}\,\mathtt{bool}; \mathsf{g} : \mathsf{Pre}\,\mathtt{int}\}$.

— $\{\mathtt{e} : \mathsf{Pre}\,\mathtt{bool}; \mathtt{f} : \mathsf{Pre}\,\mathtt{bool}; \mathtt{g} : \mathsf{Pre}\,\mathtt{bool}\} \to \{\mathtt{e} : \mathsf{Abs}; \mathtt{f} : \mathsf{Abs}; \mathtt{g} : \mathsf{Abs}\}$ est le type d'une fonction prenant en argument un enregistrement contenant 3 booléens et retournant un enregistrement vide.

Les rangées

Les rangées (ou *rows*) [153][42] ajoutent aux enregistrements les idées de modèles pour le type des champs qui ne sont pas explicitement mentionnés et la possibilité d'avoir des champs, dont la présence où l'absence n'est pas connue [168].

Intéressons-nous d'abord aux modèles : $\partial\,\mathsf{Abs}$ signifie que tous les autres champs de la rangée sont absents. $\partial\,\mathsf{Pre}\,\tau$ signifie que tous les autres champs de la rangée ont pour type τ.

La syntaxe est définie ci-dessous :

$$e ::= x \mid v \mid e\ e \qquad v ::= \lambda x :\ \tau.e \mid r \qquad \tau ::= \sigma \mid \tau \to \tau \mid \{\rho\}$$

$$\rho ::= l :\ \pi; \rho \mid \partial\pi$$

$$\pi ::= \mathsf{Pre}\ \tau \mid \mathsf{Abs}$$

La règle de typage est très proche de la règle pour les enregistrements :

$$\textsc{TRow}\ \frac{L \subset \mathbb{L} \qquad \forall l \in L, \Gamma \vdash a_l :\ \tau_l}{\Gamma \vdash \{(l = a_l)_{l \in L}\} : \{(l : \mathsf{Pre}\,\tau_l)_{l \in L}; \partial\,\mathsf{Abs}\}}$$

Par exemple : le type $\{a : \mathsf{Pre}\,\mathit{bool}; \partial\,\mathsf{Abs}\}$ est le type d'une rangée contenant un champ a de type booléen et les autres champs sont absents. Le type d'une rangée avec tous les champs présents et de type `int` est $\{\partial\,\mathsf{Pre}\,\mathtt{int}\}$.

On remarquera que $\partial\,\mathsf{Abs}$ et $\partial\,\mathsf{Pre}$ sont absorbants :

$$\{l : \mathsf{Abs}; \partial\,\mathsf{Abs}\} = \{\partial\,\mathsf{Abs}\}$$

$$\{l : \mathsf{Pre}\,\tau; \partial\,\mathsf{Pre}\,\tau\} = \{\partial\,\mathsf{Pre}\,\tau\}$$

Un même type peut donc avoir plusieurs écritures différentes. On peut cependant définir une forme normale pour écrire tous les types sous la forme :

$$\underset{i\in I}{;}\ a_i : \pi_i \,;\partial\pi \qquad \text{avec } \forall i \in I, \pi_i \neq \pi$$

En fixant π, la présence par défaut (en choisissant $\top$ par exemple), toutes les rangées peuvent être écrites sous la même forme, ce qui simplifie l'écriture de certaines règles.

À Abs et $\mathsf{Pre}\,\tau$ on ajoute $\top$, qui signifie que les autres champs de la rangée peuvent être présents ou absents et de n'importe quel type. Par exemple, $\{a = 1; b = bool\}$ et $\{a = 0\}$ correspondent tous les deux à une rangée $\{a : \mathsf{Pre}\,int; \top\}$.

Il est également possible de définir une relation de sous-typage sur les types des rangées, que nous ne définirons pas ici. En effet, cette relation est complexe dans le cas générique présenté ici, nous nous contenterons de la définir uniquement pour l'utilisation particulière des rangées que nous ferons dans le chapitre 7.

4.3.5 Typage graduel

Principe

Il est parfois difficile de typer statiquement entièrement un programme. Par exemple, le résultat du téléchargement d'un fichier sur internet peut être soit le fichier souhaité, soit une erreur envoyée par le serveur, ce qui peut être un problème pour le système de types.

Le typage graduel [188] permet de pallier ce problème en proposant de typer statiquement les parties du programme pour lesquelles c'est possible, et dynamiquement le reste du programme. Il est donc possible de faire certaines vérifications statiques de type, tandis que les autres seront réalisées à l'exécution. L'objectif est de faire autant de vérifications statiques que possible et de limiter au maximum l'utilisation du typage dynamique.

Dans une chaîne d'obfuscation, certaines informations sont dynamiques par nature, par exemple il est impossible de savoir statiquement sur quel programme la chaîne va être exécutée. Ainsi avec le typage graduel, le typage dynamique sera utilisé pour les données dépendantes du programme cible et le type que doivent avoir ces données pour que l'obfuscation soit possible sera défini dans la partie statique. Ainsi si les conditions initiales sont vérifiées, il n'y aura aucune erreur dans l'exécution de la chaîne d'obfuscation. Le typage graduel permet d'assurer que les erreurs de type proviennent de la partie typée dynamiquement du programme [189].

4.3.6 Lambda-calcul graduellement typé

Nous allons maintenant définir un λ-calcul graduellement typé. Pour cela on définit le type dynamique ? et la relation de cohérence $\sim$ telle que :

$$\tau ::= \tau \to \tau \mid ?$$

$$\tau \sim \tau \qquad \frac{\tau \sim \tau'}{\tau' \sim \tau} \qquad \tau \sim ?$$

Ces règles signifient que la relation de cohérence $\sim$ est réflexive et symétrique. De plus, tous les types sont cohérents avec le type graduel (?).

Dans le λ-calcul, la règle de typage TApp est remplacée par les règles GApp1 et GApp2 qui permettent d'utiliser des termes typés dynamiquement :

$$\textsc{GApp1}\ \frac{\Gamma \vdash e_1 : ? \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : ?} \qquad \textsc{GApp2}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau'' \qquad \tau'' \sim \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

La règle GApp1 indique que si l'abstraction est typée dynamiquement, alors l'application est bien typée et le type résultant est dynamique. La règle GApp2 signifie qu'une application est bien typée à condition que le type de l'argument soit cohérent avec le type de l'entrée de la fonction (ils peuvent donc potentiellement être typés dynamiquement). Comme pour le type de l'application sans typage graduel (règle TApp), le type résultat de l'application est le type résultat de la fonction (τ'). Bien sûr, si un des termes possède un type dynamique, il sera peut-être nécessaire d'insérer une vérification dynamique du type, que nous allons maintenant présenter.

Langage de cast

Pour obtenir la vérification de type dynamique pour les expressions typées graduellement, on ajoute une nouvelle construction le *cast* [96].

L'opération de *cast* est notée $(< \tau >\ e)$ et signifie que l'expression e sera considérée comme de type τ pendant la compilation et son type réel sera vérifié durant l'exécution. La règle de typage correspondante est donc :

$$\textsc{Cast}\ \Gamma \vdash (< \tau >\ e) : \tau$$

Cette construction ne sera pas disponible dans le λ-calcul initial (que nous noterons λ dans la suite) et ne sera insérée que pendant la compilation du langage. Nous définissons un nouveau langage, le λ_{cast}, qui est un λ-calcul contenant des *casts*. Le λ-calcul λ sera compilé vers λ_{cast}, avec des *casts* insérés aux endroits nécessaires. Dans le λ_{cast} toutes les expressions seront bien typées, et les types graduels ? seront retirés partout où cela est possible.

La syntaxe de λ_{cast} est la même que celle de λ dans laquelle on ajoute l'opération de *cast* :

$$e_c ::= x \mid \lambda x : \tau.e_c \mid e_c\ e_c \mid (< \tau >\ e_c) \qquad v ::= \lambda x : \tau.v \mid (< \tau >\ v) \qquad \tau ::= \tau \to \tau \mid ?$$

On notera la traduction de λ vers λ_{cast} : $\lambda \Rightarrow \lambda_{cast}$. Les règles de traduction sont les suivantes :

$$\textsc{CApp1}\ \frac{\Gamma \vdash e_1 \Rightarrow e_{c1} : ? \qquad \Gamma \vdash e_2 \Rightarrow e_{c2} : \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow (< \tau \to ? >\ e_{c1})\ e_{c2} : ?}$$

$$\textsc{CApp2}\ \frac{\Gamma \vdash e_1 \Rightarrow e_{c1} : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e_{c2} : \tau'' \qquad \tau'' \neq \tau \qquad \tau'' \sim \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e_{c1}\ ((< \tau >\ e_{c2})) : \tau'}$$

$$\textsc{CApp3}\ \frac{\Gamma \vdash e_1 \Rightarrow e_{c1} : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e_{c2} : \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e_{c1}\ e_{c2} : \tau'} \qquad \textsc{CVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow x : \tau}$$

$$\textsc{CLam}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow e_c : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e \Rightarrow \lambda x : \tau_1.e_c : \tau_1 \to \tau_2}$$

Les règles CApp1 et CApp2 indiquent que si le type de l'un des termes de l'application est dynamique, alors il faut insérer un *cast* sur ce terme pour vérifier dynamiquement que son type réel vérifie les règles de typage λ. Les autres règles (CApp3, CVar et CLam)

indiquent simplement que la traduction de ces termes de λ vers λ_{cast} est directe. Elles typent également ces termes en respectant les règles du λ-calcul simplement typé.

La sémantique de λ_{cast} est complètement identique à celle de λ à l'exception que l'on ajoute une règle pour le cast.

$$\textsc{Cast}\ \frac{e_c \Downarrow v \qquad v : \tau}{(<\tau>\ e_c) \Downarrow v}$$

$$\textsc{CastErr}\ \frac{e_c \Downarrow v \qquad v : \tau' \qquad \tau' \neq \tau}{(<\tau>\ e_c) \Downarrow \textsc{Error}}$$

Si, à l'exécution, le type de la valeur correspond au type dans lequel l'expression est *cast*, alors l'expression s'évalue bien dans la valeur, sinon elle s'évalue en une erreur.

Exemple

Prenons un programme qui prend en argument deux paramètres : une fonction quelconque f et un autre argument x, et applique la fonction sur l'argument $(f\ x)$. Ces arguments ne seront connus que dynamiquement (et leur type réel également), ils seront donc typés ? dans le programme.

On peut définir ce programme dans λ comme suit :

$$\lambda f : ?.\lambda x : ?.f\ x$$

Grâce aux règles de traduction décrites précédemment, on peut traduire ce programme vers λ_{cast}. On obtient le programme suivant :

$$\lambda f : ?.\lambda x : ?.(<? \to ? >\ f)\ x$$

On remarque qu'un *cast* a été inséré avant l'application de la fonction. Ce *cast* vérifiera dynamiquement que le premier argument du programme est bien une fonction, dans le cas contraire, le programme résultera en une erreur à l'exécution.

4.4 λ-calcul à deux étapes

4.4.1 Principe

De la même façon qu'il peut être intéressant d'utiliser le typage graduel pour calculer le type d'un programme statiquement lorsque c'est possible et dynamiquement lorsque cela n'est pas possible, on peut également évaluer un programme en plusieurs étapes : une étape dite « statique » et une étape dite « dynamique ». Cela est particulièrement utile sur les programmes qui s'exécutent avec des paramètres donnés dynamiquement en entrée. En effet, dans ce cas là, on ne peut pas (en général), évaluer statiquement entièrement le programme, car l'évaluation de certains termes va dépendre du paramètre d'entrée qui est dynamique. Cependant, ces programmes contiennent souvent des portions de code qui sont indépendantes du paramètre d'entrée. Évaluer statiquement ces portions a plusieurs bénéfices : cela permet d'éviter certains calculs dynamiques et donc d'obtenir une exécution plus rapide du programme [118]. Cela permet également de produire des programmes spécialisés à une utilisation et possiblement de détecter certaines erreurs avant l'exécution (la portion de code peut être statiquement évaluée en une erreur) [37].

Prenons un exemple trivial pour montrer l'application de ce type d'évaluation. Soit f la fonction :

$$f : x \to 4\frac{x}{2}$$

x est ici la variable dynamique de notre programme. On ne peut pas calculer la valeur du programme $f(x)$ sans connaitre la valeur de x. Cependant, on peut trivialement réduire le calcul de cette fonction au calcul suivant :

$$f : x \to 2x$$

Il existe plusieurs sémantiques de langages utilisant l'évaluation partielle ou les λ-calcul à deux étapes [178][93][9]. Nous allons maintenant présenter brièvement la syntaxe et la sémantique d'un λ-calcul à 2 étapes basé sur [93], sans rentrer dans les détails et l'ensemble des règles de sémantique et de typage. Car elles sont complexes et ne seront pas utilisées directement dans la suite de cette thèse.

4.4.2 Syntaxe

Les deux étapes du λ-calcul sont représentés par deux langages, $\lambda_{\mathbb{1}}$, le langage « statique », et λ_2 le langage, « dynamique ». Étant donné que la majorité des termes vont être identiques entre les deux langages, nous allons indicer les termes pour indiquer à quels langages (ou à quel « monde ») ils appartiennent. Les termes du langage statique seront indicés $\mathbb{1}$ et les termes du langage dynamique $\mathbb{2}$. Pour les règles qui ne dépendent pas du monde, nous indicerons les termes avec une variable w pour *world*.

Une syntaxe possible d'un λ-calcul à deux étapes est la suivante :

$$\mathbb{W} :: w ::= \mathbb{1} \mid \mathbb{2} \qquad \tau_w ::= \tau_w \to \tau_w \qquad \tau_{\mathbb{1}} ::= \tau_{\mathbb{1}} \to \tau_{\mathbb{1}} \mid \bigcirc\tau \qquad \Gamma ::= \bullet \mid \Gamma, x_w \,:\, \tau_w \,@\, w$$

$$e_w ::= e_w\ e_w \qquad e_{\mathbb{1}} ::= v \mid \texttt{next}(e_{\mathbb{2}}) \qquad e_{\mathbb{2}} ::= q \mid \texttt{prev}(e_{\mathbb{1}}) \mid \lambda x_{\mathbb{2}}.e_{\mathbb{2}}$$

$$v ::= x_{\mathbb{1}} \mid \lambda x_{\mathbb{1}}.e_{\mathbb{1}} \mid \texttt{next}(v) \qquad q ::= x_{\mathbb{2}} \mid \lambda x_{\mathbb{2}}.q \mid q\ q$$

La plupart des termes sont communs avec les λ-calculs usuels. Examinons plus en détail les termes spécifiques à ce langage. D'abord, le type $\bigcirc\tau$ indique que le terme aura dans le « futur » (dans l'exécution dynamique) le type τ car le terme n'est pas encore déterminé dans le monde $\mathbb{1}$. Dans le contexte de typage Γ, nous précisons dans quel monde l'association de type se trouve. Enfin les termes `next`() et `prev`() indiquent que les termes seront calculés dans $\mathbb{2}$ (pour `next`()) ou dans $\mathbb{1}$ (`prev`()).

Autrement dit, on considère ici que le développeur indique dans le programme quels sont les termes qui doivent être évalués statiquement et ceux qui seront évalués dynamiquement (par les termes `next`() et `prev`()). Ces annotations peuvent également être faites automatiquement selon le langage par une étape de *binding time analysis* [80][155][74].

4.4.3 Systèmes de type

Les règles de typage des constructions standards sont similaires à celles du λ-calcul simplement typé. Prenons par exemple, la règle de typage de l'application :

$$\textsc{TApp}\ \frac{\Gamma \vdash e_1 \,:\, \tau \to \tau' \,@\, w \qquad \Gamma \vdash e_2 \,:\, \tau \,@\, w}{\Gamma \vdash e_1\ e_2 \,:\, \tau' \,@\, w}$$

Il y a, par contre, de nouvelles règles pour les opérations `next`() et `prev`(). Ces opérations permettent de passer d'un monde à l'autre, et donc la règle TNext indique que si un terme est de type τ dans le monde $\mathbb{2}$ alors `next`(e) est de type $\bigcirc\tau$ dans le monde $\mathbb{1}$. Autrement dit, e sera « dans le futur » (dans $\mathbb{2}$) de type τ, qui est la condition de la règle. La règle TPrev fonctionne exactement sur le même principe dans le sens inverse.

$$\textsc{TNext}\ \frac{\Gamma \vdash e : \tau \mathbin{@} \mathbb{2}}{\Gamma \vdash \texttt{next}(e) : \bigcirc\tau \mathbin{@} \mathbb{1}} \qquad \textsc{TPrev}\ \frac{\Gamma \vdash e : \bigcirc\tau \mathbin{@} \mathbb{1}}{\Gamma \vdash \texttt{prev}(e) : \tau \mathbin{@} \mathbb{2}}$$

4.4.4 Sémantique

Comme pour les règles de typage, la majorité des règles de sémantique du langage sont similaires aux règles du λ-calcul usuel. Par exemple, la règle de l'application dans le monde statique est la suivante :

$$\textsc{App1}\ \frac{e_1 \Downarrow_{\mathbb{1}} \lambda x.v \qquad e_2 \Downarrow_{\mathbb{1}} v'}{e_1\, e_2 \Downarrow_{\mathbb{1}} [x \mapsto v']v}$$

Cependant, il existe également une règle pour le cas où l'application n'est pas évaluable statiquement :

$$\textsc{App2}\ \frac{e_1 \Downarrow_{\mathbb{1}} v \qquad e_2 \Downarrow_{\mathbb{1}} v' \qquad v \neq \lambda x.v''}{e_1\, e_2 \Downarrow_{\mathbb{1}} v\, v'}$$

Cette règle (App2) signifie que si le premier terme s'évalue en une valeur différente d'une abstraction, alors l'application ne peut pas se faire statiquement. Le résultat dans le monde $\mathbb{1}$ sera donc toujours une application. Cette évaluation sera à la charge de l'évaluation dynamique.

Il y a également deux règles simples pour l'évaluation de `next`() (Next) et `prev`()(Prev), qui permettent d'évaluer à l'intérieur du terme dans le monde correspondant :

$$\textsc{Next}\ \frac{e \Downarrow_2 e'}{\texttt{next}(e) \Downarrow_1 \texttt{next}(e')} \qquad \textsc{Prev}\ \frac{e \Downarrow_1 e'}{\texttt{prev}(e) \Downarrow_2 \texttt{prev}(e')}$$

Enfin, il y a également des règles qui permettent d'extraire les parties statiques d'un terme à l'extérieur d'un `next`() et inversement pour `prev`(). Ces règles consistent à extraire un terme en le liant à une variable à l'extérieur du `next`() et en le substituant par cette variable. Nous ne rentrerons pas dans le détail de la définition formelle des règles de *hoisting*, car elles sont relativement complexes et nous n'en aurons pas besoin dans la suite [5].

Nous rappelons que nous avons présenté ici une version incomplète et légèrement simplifiée de ce λ-calcul à deux étapes pour des raisons pédagogiques. L'objectif de cette partie était de présenter le principe global de fonctionnement de ce type de langage sans expliquer tous les détails formels avancés. La sémantique complète du langage est disponible dans [93].

4.5 Conclusion

Dans ce chapitre, nous avons défini un λ-calcul, présenté ses intérêts et utilisations possibles. Nous avons également présenté le principe d'un système de types, ses objectifs, et les bénéfices qu'il peut apporter à un langage. Nous avons présenté plusieurs systèmes de type pour le λ-calcul, en particulier nous avons décrit en détail le système de rangée qui permet de contenir plusieurs informations dans un type, et également d'indiquer que certaines informations sont inconnues. Nous avons également présenté le typage graduel, qui permet de typer un programme statiquement lorsque c'est possible, et dynamiquement lorsque cela ne l'est pas. Cela se fait en insérant des vérifications dynamiques de type, qui sont appelées *cast*, dans le programme. Enfin, nous avons également réalisé une description rapide du principe et de la sémantique d'un λ-calcul à deux étapes.

Ce sont des outils dont nous allons nous servir pour créer notre langage fonctionnel dédié à la conception et l'exécution de chaînes de compilation. La sémantique de ce langage dédié est décrite dans les chapitres 6 et 8. Le système de types du langage sera lui décrit dans le chapitre 7.

5. Elles sont disponibles dans [93].

Chapitre 5

Notre processus de compilation et d'obfuscation

5.1 Introduction

Dans les chapitres précédents, nous avons présenté les processus de compilation des programmes (Chapitre 3). Nous avons également présenté différentes techniques de protection logicielle contre les attaques MATE (chapitre 2) et nous avons montré les limitations des outils (en particulier des compilateurs) et des techniques de conception de chaînes de compilation usuelles pour l'application de ces protections.

Dans ce chapitre, nous allons présenter le fonctionnement global de notre processus de création de chaînes de compilation contenant des transformations d'obfuscation. Nous allons détailler notre définition du concept de passes de compilation et notre approche pour la conception et la composition de ces passes qui prend en compte les spécificités de l'obfuscation, et qui est également applicable à toutes les passes de transformation et d'analyse. Nous détaillerons comment cette approche offre plus d'options et de sûreté aux développeurs pour la composition de passes. En particulier, notre approche permet plus d'interactions entre les passes, en leur permettant de communiquer grâce à un système de propriétés associées aux programmes, et de paramètres d'entrées et sorties de passes plus expressifs. Nous allons utiliser cette approche dans un langage fonctionnel dédié à la conception de chaînes de compilation, pour apporter plus de sûreté dans l'application des passes de compilation.

Dans ce chapitre, les contributions principales qui seront présentées seront donc notre définition des passes de compilation et notre approche pour la conception de chaînes de compilation et d'obfuscation. Nous présenterons également dans ce chapitre les principes et le fonctionnement global de notre langage SCOL (*Safe Control of Obfuscation Language*), ainsi que le processus de conception, de compilation et d'exécution de scripts de compilation conçus avec notre approche. Les détails du langage seront donnés dans les

chapitres 6 et 7, et de son exécution dans le chapitre 8.

5.2 Passes d'obfuscation

5.2.1 Introduction

Une passe est une fonction dans un compilateur qui effectue des transformations ou des analyses sur le code, le compilateur coordonne l'application de ces passes dans le processus de compilation d'un programme. Comme nous l'avons décrit dans le chapitre 3, les transformations d'obfuscation sont souvent réalisées sous la forme de passe de compilation, mais ces passes ont des spécificités pour lesquelles les compilateurs usuels sont peu adaptés. Nous allons présenter ici notre définition d'une passe et notre approche pour la conception de passes que nous utiliserons ensuite dans notre processus de création de chaînes de compilation.

5.2.2 Contenu d'une passe

Une technique d'obfuscation est souvent composée de plusieurs transformations. Par exemple, on peut décomposer le *Bogus Control Flow* (cf 2.5.1) en trois transformations :

— Une transformation qui crée de fausses branches dans le programme
— Une transformation qui génère le code contenu dans ces fausses branches
— Une transformation qui crée des prédicats opaques pour les conditions d'entrées des fausses branches

Nous proposons de séparer cette transformation en créant des passes atomiques qui séparent ces trois transformations. D'un point de vue du génie logiciel, il est intéressant de séparer ces transformations. Certaines peuvent être réutilisées dans plusieurs transformations : les constantes opaques seront notamment utilisées dans deux nombreuses passes. De plus, pour remplir certains objectifs (par exemple créer des constantes opaques), il peut y avoir plusieurs transformations possibles. Séparer ainsi les transformations permet d'utiliser plusieurs méthodes (dans ce cas pour créer des constantes opaques) dans le processus d'obfuscation, ce qui rend le programme plus divers et donc possiblement plus difficile à attaquer.

Enfin, il est plus simple d'exprimer les effets sur le programme de passes atomiques qui sont donc plus simples. Cela peut permettre d'obtenir plus facilement des informations

sur le niveau de protection d'un programme et potentiellement de simplifier la preuve de certains comportements du processus de compilation et d'obfuscation.

5.2.3 Entrée/sortie d'une passe

En général, les paramètres d'entrée et de sortie d'une passe peuvent différer d'un compilateur à l'autre, mais toutes les passes pour un même compilateur auront la même signature (ou un choix parmi un nombre réduit de signatures) pour faciliter l'intégration de ces passes dans le gestionnaire de passes.

Par exemple, dans LLVM [135], les passes héritent de la classe abstraite `FunctionPass` (ou `ModulePass`, `LoopPass`, ...) et vont surcharger la méthode `bool RunOnFunction(Function &F)` (ou `bool RunOnModule(Module &M)`, ...). Ces méthodes prennent en paramètres une référence vers l'entité du programme (la fonction, le module ...) à modifier et retournent une variable booléenne qui indique si cette passe a effectué des modifications sur le programme.

Entrée d'une passe

Les paramètres d'entrée des passes dans LLVM limitent les possibilités dans l'écriture des passes d'obfuscation. On ne peut fournir à la passe que l'entité que l'on souhaite modifier et aucun autre paramètre permettant de choisir les options d'applications de la passe.

Par exemple, il n'est pas possible de choisir la valeur d'une graine de générateurs aléatoire dans les cas où la passe utilise de l'aléatoire. Cela rend la reproductibilité de l'application de la passe difficile. De même il n'est pas possible de fournir à la transformation de *bogus control flow* (cf 2.5.1) la méthode à utiliser pour générer les prédicats opaques.

Il est bien sûr possible de contourner ce problème sans changer le fonctionnement et la structure des passes et du gestionnaire de passes, par exemple en utilisant les attributs de l'instance de l'objet correspondant à la passe. Cependant, cela complexifie l'écriture et la lecture du script d'application des passes, en particulier si les passes sont utilisées plusieurs fois avec des paramètres différents, et demande des outils supplémentaires dans le gestionnaire de passes pour appeler ces méthodes de paramétrisation (normalement, le gestionnaire de passes se charge juste d'appeler des passes et des analyses).

Dans notre processus d'obfuscation, la signature d'une passe est libre et choisie par le

développeur de la passe. Elle contiendra l'ensemble des informations nécessaires à l'exécution de la passe, dont notamment les graines de générateurs aléatoires et les autres paramètres de la passe. Dans notre approche, nous recommandons de permettre le choix de tous les paramètres influant sur l'exécution de la passe à l'utilisateur et éventuellement de fournir des fonctions *wrapper* pour les utilisateurs non experts. Notre gestionnaire de passes a été adapté pour utiliser des passes à signatures variables.

Sortie d'une passe

Dans les compilateurs traditionnels, les passes ont accès directement au code des fonctions (ou une autre entité sur laquelle elles s'appliquent), par exemple dans LLVM via les références qui sont passées en paramètre de la méthode `runOnFunction`. Les transformations sur ces fonctions sont réalisées par effet de bord de la méthode `runOnFunction`. Dans LLVM, le paramètre de sortie est un simple booléen qui indique si le programme a été modifié par la passe, ce qui permet au gestionnaire de passes de savoir s'il est nécessaire de recalculer certaines analyses.

Dans notre processus, nous utilisons une approche fonctionnelle des passes : les modifications sur le code d'une fonction ne sont pas faites par effet de bord dans la passe ; la passe prend en paramètre la fonction originale et retourne la fonction transformée. Dans le processus de compilation, la fonction originale peut ensuite être remplacée par la fonction transformée si nécessaire. La passe pourra aussi retourner en plus certaines informations potentiellement utiles à la composition de plusieurs passes, par exemple la liste des constantes opaques créées par le *bogus control flow*. Comme pour les entrées, le choix des retours de la passe est donc laissé au développeur de celle-ci.

Cette approche a plusieurs avantages :

— Il est possible de caractériser l'effet d'une transformation via son type. Il y a donc moins de risques d'erreur et d'incompatibilité lors de la composition de transformations. Nous détaillerons comment nous allons réaliser cela dans le chapitre 6 et surtout le chapitre 7.

— Il est possible de réutiliser le résultat d'une passe en entrée d'une autre passe : cela facilite la composition de passes.

— Les passes donnent des informations plus précises sur leur fonctionnement, ce qui permet une description plus fine du processus d'application des passes dans le gestionnaire de passes.

Les défauts de cette approche sont qu'à l'exécution de la chaîne de compilation, de

nombreuses copies des fonctions transformées peuvent être créées avec une approche naïve, ce qui est peu efficace au niveau de la vitesse d'exécution et de la taille mémoire. Cependant, certaines optimisations permettent de réduire ce problème. De plus, comme les transformations n'ont plus de signatures communes, l'écriture du code d'application de ces passes est plus complexe. Ce qui peut être réglé avec des outils pour automatiser ou faciliter l'écriture du script de compilation tel que le langage présenté dans le chapitre 6.

5.3 Composition de passes

5.3.1 Introduction

Nous avons vu dans le chapitre précédent que la composition de passes d'obfuscation était plus complexe que celles d'optimisation et que les gestionnaires de passe étaient peu adaptés à l'obfuscation. Pour résoudre ces problèmes de composition, nous proposons d'ajouter certaines informations dans les passes et le gestionnaire de passes. Dans cette section, nous allons présenter le système de propriétés qui contient ces informations et comment celui-ci peut être utilisé par les différentes passes et le gestionnaire de passes pour résoudre les problématiques de composition de passes.

5.3.2 Propriétés

Dans notre processus de compilation, chaque fonction du programme à compiler possède un ensemble de propriétés. Ces propriétés contiennent toutes les informations nécessaires au bon déroulement de la compilation, en particulier de l'obfuscation et de l'optimisation.

Les propriétés sont des valeurs associées aux fonctions. La liste des propriétés n'est pas fixe, mais peut être fixée par les développeurs de passes et du gestionnaire de passes selon les besoins pour la chaîne de compilation, et une fonction peut avoir un nombre quelconque de propriétés. Une propriété peut être associée à une fonction depuis plusieurs sources :

— par le développeur du programme, notamment grâce a des annotations dans le code source.
— par les passes d'obfuscation et d'optimisation qui peuvent ainsi indiquer des informations sur les modifications qu'elles ont effectuées.
— par des passes d'analyses. Certaines propriétés peuvent être calculées et peuvent donc être associées aux fonctions via les analyses.

5.3.3 Préconditions d'application de passes

Certaines passes ont besoin que le programme soit dans un certain état pour être appliquées (cf 3.3). Dans notre processus, ces préconditions à l'application de passes vont être représentées par les propriétés associées aux fonctions.

Il y a trois types de préconditions possibles :

— les prérequis : si une passe a besoin que le code sur lequel elle s'applique ait certaines propriétés obligatoirement.
— les interdits : lorsqu'une passe ne peut pas s'appliquer si la cible possède certaines propriétés.
— les suggestions : certaines transformations peuvent suggérer que certaines propriétés soient présentes/absentes dans la cible pour une meilleure efficacité, mais fonctionner correctement même si ce n'est pas le cas. Dans le cas où les suggestions ne sont pas respectées, la passe ou le gestionnaire de passes pourront donner un avertissement à l'utilisateur.

On pourra définir ces préconditions dans la définition des passes et utiliser notre gestionnaire de passes pour vérifier que ces conditions sont respectées. Les détails de la gestion des conditions sont décrits dans le chapitre 7.

5.3.4 Postconditions à l'application de passes

Certaines passes requièrent des transformations supplémentaires, ou interdisent l'application d'autres transformations, après leur application. Par exemple, en général, on ne veut pas appliquer de transformation après avoir ajouté du *tamperproofing* au programme, car cela peut empêcher les fonctions de détection de fonctionner (cf. 3.3.3). Comme pour les préconditions, on peut distinguer les postconditions dans les trois mêmes catégories : les applications obligatoires, les interdictions d'applications, et les suggestions de transformations à appliquer ensuite.

De la même façon que pour les préconditions, on peut utiliser les propriétés pour gérer les postconditions des passes. On pourra indiquer dans la définition des passes quelles sont les postconditions, et les passes ajouteront des propriétés après leurs applications pour que le gestionnaire de passes puisse gérer ces conditions.

```
int foo(int);
int bar(int);

int foobar(int a){
    int r = foo(a);
    return bar(r);
}
```

FIGURE 5.1 – Un programme simple

5.3.5 Interactions entre les passes (transmissions d'informations)

Dans notre processus, il y a plusieurs moyens d'obtenir des interactions entre les passes. Comme les paramètres d'entrée et de sortie d'une passe peuvent être choisis par le développeur de la passe (et ne sont pas fixés par le gestionnaire de passes comme dans la majorité des compilateurs), ils peuvent être utilisés pour transmettre des informations entre les passes lors de leur composition. Les passes peuvent utiliser les sorties d'autres passes comme paramètres d'entrée. Par exemple, une passe de *bogus control flow* (cf. 2.5.1) pourra transmettre la liste des prédicats servant à masquer le vrai flot de contrôle à une transformation de prédicats opaques qui les rendra opaques.

Une autre possibilité est d'utiliser les propriétés : une passe peut donner certaines propriétés à une fonction, les passes suivantes peuvent avoir un comportement différent en fonction de ces propriétés. Par exemple, une transformation de *control flow graph flattening* peut indiquer avec des propriétés que le bloc de *dispatch* est critique et doit être obfusqué « fortement ».

Selon le comportement souhaité, l'une ou l'autre de ces méthodes pourra être plus ou moins pratique et facile à implémenter.

5.4 Exemple de scénario de compilation

Prenons l'exemple du programme en figure 5.1 qui définit une fonction `foobar` qui enchaîne deux fonctions `foo` puis `bar`. Ce programme peut être obfusqué en lui applicant successivement plusieurs transformations afin de maximiser l'efficacité de la protection. Par exemple, la chaîne de compilation que nous allons appliquer est représentée en figure 5.2. Tout en restant simple, elle n'utilise que trois obfuscations, elle contient déjà un peu de complexité. Pour que la protection soit efficace, les transformations doivent être

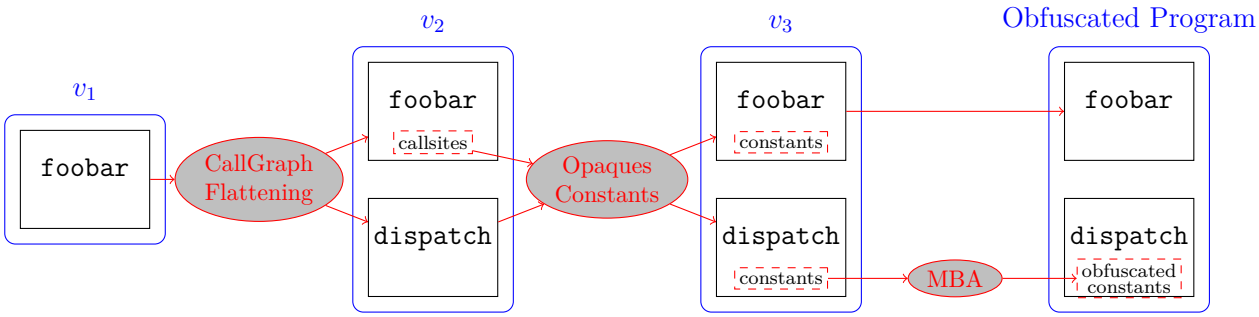

FIGURE 5.2 – Un exemple de chaîne d'obfuscation

appliquées dans le bon ordre, et certaines de ces transformations utilisent le résultat des transformations précédentes.

Les trois transformations appliquées ici pour l'exemple au niveau du code source donneraient les programmes des figures 5.3, 5.4 et 5.5. Ces transformations sont usuellement réalisées sur la représentation intermédiaire LLVM, mais sont représentées dans ces figures sur du code C pour plus de lisibilité.

Dans un premier temps, on applique la transformation de *CallGraph Flattening* (figure 5.3) qui transforme tous les appels de fonction en appel à une fonction `dispatch` qui renvoie la bonne fonction.

Ensuite, on applique une transformation de *constantes opaques* (figure 5.4) qui transforme les constantes servant à identifier quelles sont les fonctions à appeler dans le `dispatch` et dans `foobar` en expression qui apparaissent comme dépendantes du contexte (mais qui reste constante, peu importe le contexte).

Enfin, pour rendre plus difficile l'identification et le calcul de ces constantes opaques, et pour qu'elles ne soient pas calculées avec la même expression dans `dispatch` et `foobar`, on applique une transformation de MBA (*Mixed Boolean Arithmetic*) (figure 5.5) sur `dispatch` qui rend l'expression de calcul de la constante plus difficile à simplifier par un solveur. En particulier, cela permet d'éviter que les optimisations appliquées potentiellement après ces transformations retirent la protection, il s'agit d'un comportement similaire à celui que nous avons décrit dans la figure 3.2.

La constante opaque et le calcul en MBA utilisés ici sont inspirés de [92].

```
int foo(int);
int bar(int);

void* dispatch(unsigned int state) {
    if (state == 0)
        return foo;
    else if (state == 1)
        return bar;
}

int foobar(int a){
    int r = ((int (*)(int)) dispatch(0))(a);
    return ((int (*)(int)) dispatch(1))(r);
}
```

FIGURE 5.3 – Exemple d'application du *CallGraph Flattening*

```
int foo(int);
int bar(int);

void* dispatch(unsigned int state) {
    int C1 = (((state - 2) | (1 - state)) + 1);/* C1 = 0 */
    int C2 = /* ... */;
    if (state == C1)
        return foo;
    else if (state == C2)
        return bar;
}

int foobar(int a){
    int C1 = (((a - 2) | (1 - a)) + 1);/* C1 = 0 */
    int C2 = (x * x + x + 1) % 2; /* C2 = 1 */
    int r = ((int (*)(int)) dispatch(C1)();
    return ((int (*)(int)) dispatch(C2)(r);
}
```

FIGURE 5.4 – Exemple d'application des *constantes opaques*

```
int foo(int);
int bar(int);

void* dispatch(unsigned int state) {
    int C1 = (((state - 2) & ~(1 - state)) + (1 - state) + 1;/* C1 = 0 */
    int C2 = /* ... */;
    if (state == C1)
        return foo;
    else if (state == C2)
        return bar;
}

int foobar(int a){
    int C1 = (((a - 2) | (1 - a)) + 1);/* C1 = 0 */
    int C2 = /* ... */;
    int r = ((int (*)(int)) dispatch(C1)();
    return ((int (*)(int)) dispatch(C2)(r);
}
```

FIGURE 5.5 – Exemple d'application de la *Mixed Boolean Arithmetic*

5.5 Processus de compilation

5.5.1 Introduction

Dans ce chapitre nous avons déjà décrit comment les passes sont définies, quels types de transformations elles peuvent appliquer, sur quelles cibles et quels sont leurs paramètres d'entrées et de sorties.

Notre processus d'obfuscation se base sur l'infrastructure de compilation LLVM [136], en particulier son gestionnaire de passes. Nous changeons le comportement du gestionnaire de passes de LLVM pour gérer les passes d'obfuscation. Cela signifie aussi que nous nous limitons aux transformations exécutées sur la représentation intermédiaire de LLVM. Nous avons choisi l'infrastructure de compilation de LLVM, car il s'agit d'une infrastructure de compilation couramment utilisée industriellement et que de nombreuses passes d'obfuscation sont implémentées avec LLVM dans la littérature et dans des obfuscateurs industriels. De plus, une grande partie des transformations d'obfuscation peuvent être implémentées pour être appliquées à la compilation (cf. 2.6), donc dans LLVM. Cependant, ce choix nous empêche d'utiliser des transformations qui ne peuvent pas être implémen-

tées à la compilation, en particulier celles qui doivent s'exécuter sur le code exécutable. Enfin, notre processus ne peut s'appliquer que sur les programmes écrits dans les langages qui sont gérés par l'infrastructure de compilation LLVM (C, C++, Objective-C ...). Il est possible d'appliquer les résultats de cette thèse dans d'autres étapes de la compilation et dans d'autres infrastructures avec plus ou moins de modifications (plus de détails en chapitre 9).

Nous allons maintenant nous intéresser au processus global de la compilation d'un programme avec notre approche. Le processus complet est décrit dans la figure 5.6. Les détails techniques sont donnés dans les chapitres suivants.

5.5.2 Script d'obfuscation

Le processus de compilation est décrit dans un **script d'obfuscation** dans un langage dédié : SCOL (*Safe Control of Obfuscation Language*). Le script d'obfuscation décrit l'ensemble de la chaîne de compilation avec une granularité fine, plus de dynamisme et de vérification que dans un gestionnaire de passes traditionnel.

L'utilisation d'un langage dédié apporte plusieurs avantages. D'abord, cela permet aux développeurs de scripts d'obfuscation de manipuler des abstractions de haut niveau pour les concepts des chaînes de compilation (en particulier, les passes et les programmes), telles que nous les avons définis au début de ce chapitre. Le développeur de script d'obfuscation n'a donc pas besoin de connaître le fonctionnement interne du compilateur et des différentes passes de compilation. De plus, un langage dédié offre également plus d'expressivité que certaines autres techniques de contrôle de l'application des passes (par exemple, l'utilisation d'option de compilation comme `-O2`). Enfin, la création d'un langage dédié permet également l'utilisation d'un système de types spécifique à ce langage qui fournit des avantages supplémentaires que nous détaillerons en 5.5.3. Nous détaillerons et justifierons ces différents avantages de façon plus complète dans les chapitres suivants.

On souhaite faire une évaluation partielle du script d'obfuscation avant son exécution, c'est-à-dire calculer une partie des expressions d'un script pendant la compilation du script. Cela permet de réduire le temps d'exécution du programme en contrepartie d'un temps de compilation plus long, car une partie des calculs est réalisée à la compilation plutôt qu'à l'exécution. Nous n'avons cependant pas mesuré l'impact de cette évaluation partielle sur les performances de la chaîne de compilation.

Pour cela la compilation du langage est réalisée en plusieurs étapes. Dans un premier temps, il y a une étape de traduction qui évalue les termes qui peuvent être calculés stati-

quement dans le script (c'est-à-dire avant l'exécution). Cette étape génère un script dans un langage intermédiaire, que nous appellerons SCOL dynamique, car il ne contient plus que des termes qui ne sont pas calculables statiquement. Par opposition, nous appellerons le premier langage (celui qui contient encore des termes calculables statiquement), le SCOL statique. (cf. Fig. 5.6).

Les travaux de cette thèse ont été réalisés avec un objectif de généricité dans les cibles (compilateurs divers ou autre outil d'obfuscation), bien que les résultats pratiques soient concentrés sur LLVM. L'utilisation de langages intermédiaire devrait faciliter l'extension de notre processus à d'autres cibles. L'utilisation de langages intermédiaires nous semblait également faciliter l'implémentation de ce processus.

Les détails du fonctionnement du langage et de la traduction vers le langage dynamique sont donnés dans le chapitre 6.

5.5.3 Système de types

Nous avons défini un système de types pour notre langage dédié à la création de scripts d'obfuscation. Ce système de types permet de détecter certaines erreurs, et il contient également les informations sur les propriétés que nous avons décrites en 5.3.2. Nous souhaitons notamment assurer que les contraintes d'applications des passes sont respectées.

Dans notre système de types, nous allons utiliser le typage graduel (cf. 4.3.5), pour typer statiquement les scripts lorsque cela est possible et dynamiquement sinon. En effet, les scripts d'obfuscation sont appliqués sur des programmes et ces programmes (et leurs types) ne sont pas connus avant l'exécution. Par contre, on peut faire de nombreuses vérifications de type simplement avec les parties typable statiquement (notamment vérifier que certains enchaînements de passes sont possibles). Autrement dit, on cherche à savoir si une composition de passes est toujours possible (le script est statiquement bien typé sans vérification dynamique), toujours impossible (le script est statiquement mal typé) ou si la composition est possible dans certains cas (script bien typé contenant des vérifications dynamiques).

L'utilisation du typage graduel ajoute une étape au début du processus de compilation des scripts d'obfuscation qui est l'insertion des *cast*. La première étape de la compilation du script est donc la vérification des types et l'insertion des *casts* qui traduit le script initial vers le langage statique. (cf. Fig. 5.6)

Le système de types de notre langage est détaillé dans le chapitre 7.

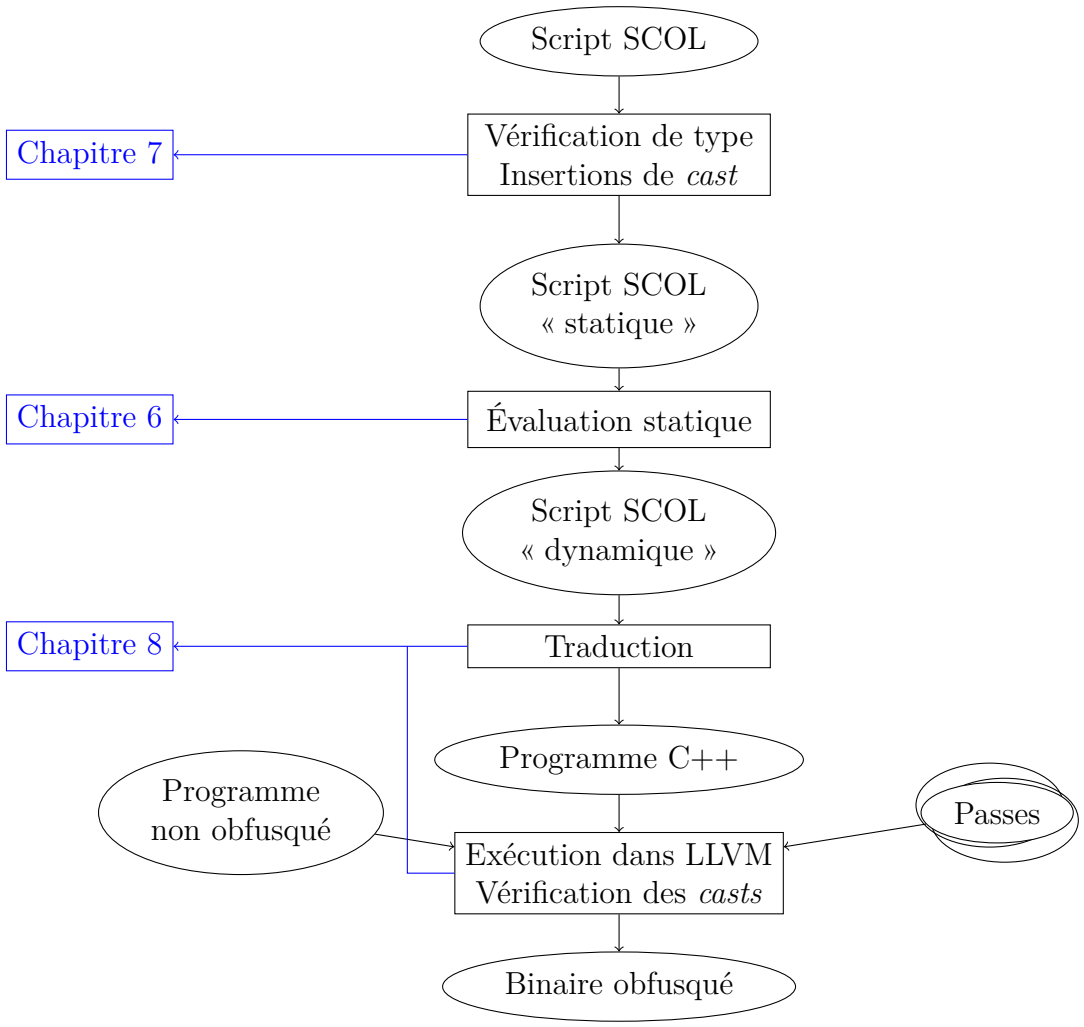

FIGURE 5.6 – Processus de compilation et d'exécution d'un script SCOL

5.5.4 Compilation et exécution du script d'obfuscation

Dans notre processus d'obfuscation, le script d'obfuscation est compilé vers du code C++ qui représente notre version du gestionnaire de passes LLVM. Le programme cible est compilé depuis un des langages sources possédant un *front end* pour LLVM vers le langage intermédiaire (IR) LLVM. Les passes sont exécutées par notre gestionnaire de passes conformément au contenu du script d'obfuscation sur la représentation intermédiaire du programme cible. Enfin le programme est compilé vers l'une des architectures cibles disponibles dans LLVM, pour générer le binaire final protégé. (cf. Fig. 5.6) Les détails de ce processus de compilation sont présentés dans le chapitre 8.

5.5.5 Prototype

Pour tester notre processus en pratique, nous avons réalisé un prototype basé sur notre approche. Ce prototype comprend un compilateur de SCOL. Ce compilateur est développé en OCaml, il prend en entrée un script SCOL et génère en sortie du code C++ correspondant au gestionnaire de passes appliquant la chaîne de compilation souhaitée. Ce compilateur effectue les différentes vérifications (particulièrement de type) et les étapes de traduction qui sont décrites dans ce chapitre et dans les chapitres 6, 7 et 8. Ce gestionnaire de passe est ensuite compilé comme une passe (au sens de LLVM) dynamique. Elle peut ensuite être chargée par les outils LLVM (comme CLang et opt) avec les options (déjà intégrées à ces outils) qui permettent de charger et d'exécuter des passes dynamiques. Donc, pour compiler un programme en utilisant une chaîne de compilation créée avec SCOL, il suffit d'ajouter une option de compilation aux processus de compilation existants.

Notre prototype comprend également une bibliothèque C++ qui implémente les différents concepts de notre processus dans LLVM afin de les utiliser dans notre gestionnaire de passes. Les détails de cette bibliothèque et de l'exécution du gestionnaire de passe sont donnés dans le chapitre 8.

5.6 Conclusion

Nous avons détaillé nos définitions pour les concepts fondamentaux de notre processus de compilation et notre approche pour la conception de passes et la composition de ces passes. Les spécificités que nous avons intégrées à ce processus permettent d'obtenir des avantages dans la conception des chaînes d'obfuscation par rapport aux passes usuelles. En particulier, cela permet plus de liberté et de sûreté dans la conception de script d'obfuscation. En effet, notre approche permet notamment plus de finesse dans l'application des passes sur un programme, plus d'interaction entre les passes lors de la composition et plus de sûreté dans leurs applications avec, notamment, la vérification de pré et post-conditions.

Nous avons également décrit un exemple de chaîne d'obfuscation avec notre approche, que nous allons réutiliser dans les chapitres suivants pour illustrer les différents aspects de la conception et l'exécution d'un tel script d'obfuscation. Enfin, nous avons présenter globalement notre processus de compilation et d'exécution d'un script d'obfuscation, dont les différentes étapes vont être détaillées dans les chapitres 6, 7 et 8.

Chapitre 6

SCOL, un langage pour décrire et exécuter une chaîne de compilation et d'obfuscation

6.1 Introduction

Dans les chapitres précédents, nous avons présenté les spécificités de la compilation pour l'obfuscation de programmes et les limites des processus de compilation traditionnels. Nous avons également défini notre processus de compilation et d'obfuscation de programmes prenant en compte ces particularités. Ce processus utilise plusieurs outils qui seront présentés dans ce chapitre ainsi que les suivants.

En particulier, notre solution utilise un langage dédié à la description et la génération de chaînes de compilation. Dans ce chapitre, nous allons présenter la syntaxe et la sémantique du langage SCOL (*Safe Control of Obfuscation Language*). SCOL est un langage offrant plus d'expressivité dans la génération de chaînes de compilation, en particulier pour les chaînes de compilation qui contiennent des transformations d'obfuscation. Les scripts écrits en SCOL peuvent être compilés pour être exécutés par le gestionnaire de passes LLVM et pour appliquer la chaîne de compilation et d'obfuscation au programme cible.

La compilation et l'exécution d'un script SCOL se font en plusieurs étapes. Il est d'abord compilé vers un langage intermédiaire, puis il est traduit vers du code C++. Ce code C++ est exécuté dans le gestionnaire de passes de LLVM qui applique des passes sur le programme à protéger pour générer un binaire obfusqué. Ces opérations sont décrites dans le schéma en figure 5.6.

Nous verrons la première étape (la compilation vers le langage intermédiaire) dans ce chapitre et le suivant. Le reste du processus de compilation sera présenté dans le chapitre 8.

Dans un premier temps, nous présentons la syntaxe globale du langage, notamment

les constructions syntaxiques particulières liées aux concepts de l'obfuscation, dans une version simplifiée minimale du langage, c'est-à-dire une version du langage ne contenant que les concepts principaux de la chaîne de compilation : les fonctions et les passes. Nous le présentons de cette façon pour des raisons pédagogiques, pour expliquer plus simplement le cœur du langage. Nous présentons ensuite la sémantique du langage et son processus d'évaluation partielle. Dans ce chapitre, nous présentons une version non typée de SCOL pour des raisons pédagogiques. Le système de types du langage sera décrit dans le chapitre suivant.

Dans une seconde partie, nous présentons des extensions à la version minimale du langage. Nous ajouterons notamment des données supplémentaires (entiers et booléens), des structures de données (listes et tuples), les structures conditionnelles, la récursion et des termes permettant de manipuler ces valeurs. Ces extensions permettent de réaliser des chaînes de compilation plus complexes et offrent plus de possibilités aux développeurs de scripts d'obfuscation en SCOL.

Enfin, nous présentons quelques exemples de scripts SCOL complets qui démontrent les possibilités du langage dans l'écriture de scripts d'obfuscation.

6.2 Concepts fondamentaux du langage

6.2.1 Intégration des concepts liés à l'obfuscation

Pour spécialiser notre langage pour la description et l'exécution de chaînes de compilation et d'obfuscation, nous nous sommes basés sur la structure du λ-calcul auquel nous avons ajouté plusieurs constructions. Nous avons également modifié la sémantique de certaines expressions pour correspondre à notre processus de compilation. Nous allons maintenant présenter ces différents ajouts et modifications.

Construction pour les passes

La première construction que nous ajoutons au langage est une construction permettant de manipuler les passes de compilation, incluant les passes d'obfuscation, les passes d'optimisation et les passes d'analyse. Cette construction ne décrit pas les modifications que la passe effectue sur le programme (son implémentation), mais uniquement son nom et le lieu où se trouve son l'implémentation pour le compilateur [1]. En effet, la passe est

1. Dans le chapitre suivant, nous ajouterons également son type.

implémentée à l'extérieur du langage : dans le prototype, il s'agira d'une fonction C++.

La définition d'une nouvelle passe se fait ainsi :

$$\texttt{let\_pass}\ x\ \texttt{from}\ \text{PASS}\ \texttt{in}\ e$$

La variable x est liée à la passe dans l'expression e. PASS indique où l'implémentation de la transformation se trouve. Cette information est passée au gestionnaire de passes pour l'exécution du processus de compilation et d'obfuscation, et n'est pas utilisée autrement dans SCOL.

Cette construction est très similaire dans sa syntaxe et sa sémantique à la construction `let`, à la différence que `let_pass` est uniquement la déclaration de l'abstraction et ne contient pas sa définition (qui est externe au langage).

Structure d'un script SCOL

La construction principale d'un script SCOL est $\texttt{script}(x) = e$. Informellement, un script SCOL est une fonction qui sera appliquée sur des paramètres d'entrées (qui peuvent être un programme, une liste de programmes ...). L'entrée d'un script d'obfuscation sera souvent le programme cible et la sortie sera le programme transformé par l'application du processus de passes. Le script décrit une fonction qui transforme un programme non protégé en programme protégé.

La syntaxe d'un script SCOL est la suivante :

$$\begin{aligned} \mathit{main} &::= \texttt{script}(x) = e \\ e &::= x \mid \lambda x.e \mid e\ @\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let\_pass}\ x\ \texttt{from}\ \text{PASS}\ \texttt{in}\ e \end{aligned}$$

Un script SCOL sera donc composé d'une expression *main*, qui est une abstraction, qui va contenir elle-même une suite d'abstractions et d'applications, c'est-à-dire qu'un script SCOL sera une application successive de passes de compilation sur un programme, ce qui permet de définir un processus d'obfuscation basique (un enchaînement de passes dans un ordre fixé), les extensions que nous ajouterons au langage dans la suite de ce chapitre permettront de réaliser des processus d'obfuscation plus complexes.

Notons que nous utilisons ici le mot « programme » dans un sens très général, il peut s'agir de programmes complets, de modules ou encore de fonctions selon l'implémentation de la traduction dans le gestionnaire de passes. Dans notre cas, un programme désignera toujours une fonction du programme à obfusquer. Dans le futur, l'implémentation pourrait

être étendue pour permettre de manipuler différents types d'entités.

Évaluation statique et évaluation dynamique

Une partie de l'évaluation d'un script peut se faire statiquement. Le reste de l'évaluation se fera lors de l'exécution du script d'obfuscation. On ne peut pas appliquer de passes sur un programme avant que le programme ne soit connu, ces applications devront donc être réalisées à l'exécution. Par contre, certains calculs, en particulier les calculs qui ne dépendent pas des programmes d'entrées ou de résultats d'application des passes, peuvent être évalués statiquement. Pour cela, nous nous inspirons du fonctionnement du langage en deux étapes décrites en 4.4, c'est-à-dire qu'il y aura première étape d'évaluation statique avant l'évaluation dynamique. Nous décrirons donc deux langages, un premier langage « statique » et un langage « dynamique ». Le langage dynamique est interne, il est utilisé pour notre processus d'évaluation et de compilation, mais ne devrait normalement pas être manipulé par l'utilisateur final. Les termes du langage statique sont notés e_{sta}, et ceux du langage dynamique e_{dyn}.

Certaines constructions sont communes aux langages statique et dynamique. Les constructions seront indicées sta ou dyn selon le langage auquel ils appartiennent. Par exemple, l'application dynamique est notée : $e_{dyn}@_{dyn}e_{dyn}$. Le langage auquel une construction appartient peut aussi être désigné par une variable : w.

De plus, nous créons la construction `RunPass`(PASS, e) dans le langage dynamique, qui représente l'application d'une passe (indiquée par la localisation de la passe : PASS) à un programme ou partie de programme résultat de l'expression e.

Le langage dynamique contient également le terme `pass`(PASS) qui permet de désigner une passe qui n'est (pour le moment) pas appliquée, contrairement à `RunPass`(PASS, e) qui désigne l'application de la passe.

Enfin, il existe également des valeurs particulières dans le langage qui représentent les programmes (*program*). Ces valeurs représentent les programmes au cours de l'évaluation du script SCOL : les programmes donnés en entrée du script, les programmes résultant de l'application du script et les programmes intermédiaires existants durant l'exécution du script. Ces valeurs sont abstraites du point de vue de SCOL, car il n'existe pas de terme permettant de définir ou créer un programme par un terme dans SCOL. Ils ne peuvent être représentés que sous la forme de variable ou de résultats (abstraits) de passes. Ils ne sont une valeur que pour les besoins de la présentation de la sémantique d'évaluation, pour la partie exécution du script.

La syntaxe de cette première version de SCOL est donc :

$$
\begin{array}{lll}
\mathbb{W} & ::= & \mathit{sta} \mid \mathit{dyn} \mid w \\
\mathit{main} & ::= & \texttt{script}(x) = e_{sta} \\
e_{sta} & ::= & x \mid \lambda_{sta} x.e_{sta} \mid e_{sta}\ @_{sta}\ e_{sta} \mid \texttt{let\_pass}\ x\ \texttt{from}\ \textsc{pass}\ \texttt{in}\ e_{sta} \\
e_{dyn} & ::= & \texttt{RunPass}(\textsc{pass}, e_{dyn}) \mid e_{dyn}\ @_{dyn}\ e_{dyn} \mid x \mid v \mid \textsc{Error} \\
\mathbb{V} :: v & ::= & \mathit{function} \mid \mathit{program} \\
\mathit{function} & ::= & \lambda_{dyn} x.e_{dyn} \mid \texttt{pass}(\textsc{pass})
\end{array}
$$

Nous avons distingué les fonctions des autres valeurs dans le langage. Les fonctions représentent les termes qui peuvent être appliqués, c'est-à-dire les λ-abstraction et les passes. Dans cette version minimale du langage, toutes les valeurs exprimables sont des fonctions, mais les extensions que nous décrirons dans la suite de ce chapitre ajouteront des valeurs qui ne sont pas des fonctions.

La première étape de la compilation (la transformation de terme de e_{sta} vers des termes de e_{dyn}) permet de faire certains calculs et réductions avant l'exécution du script d'obfuscation. Le script traduit et exécuté et le script original doivent donner le même résultat sémantiquement.

Dans la suite de ce chapitre, nous allons définir deux sémantiques pour notre langage : la sémantique d'évaluation et la sémantique de traduction. La sémantique d'évaluation indique le résultat final de l'exécution du script d'obfuscation sur un programme. La sémantique de traduction exprime les règles de traduction du script vers les différents langages intermédiaires (langage dynamique puis C++ dans le chapitre 8).

Exemple de programme

Maintenant que nous avons défini une syntaxe minimale pour notre langage de description de script d'obfuscation, nous pouvons commencer à écrire des scripts d'obfuscation simples.

Prenons par exemple un script qui exécute la séquence de transformations suivantes :

1. Une transformation de *bogus control flow* (cf. 2.5.1), dont la source est Obf.BCF (a fonction BCF(*Bogus Control Flow*) du module Obf (pour Obfuscation)).
2. Une transformation de *Control Flow Graph Flattening*(cf. 2.5.2), dont la source est Obf.CFGF.
3. Un ensemble d'optimisations génériques, dont la source est Opt.Generic.

Le script d'obfuscation en SCOL réalisant cette chaîne d'obfuscation sera le suivant :

$$
\begin{array}{l}
\texttt{script}(x) = (\\
\quad \texttt{let\_pass}\ \mathit{bcf}\ \texttt{from}\ \textsc{Obf.BCF}\ \texttt{in}\\
\quad \texttt{let\_pass}\ \mathit{cfgf}\ \texttt{from}\ \textsc{Obf.CFGF}\ \texttt{in}\\
\quad \texttt{let\_pass}\ \mathit{opt}\ \texttt{from}\ \textsc{Opt.Generic}\ \texttt{in}\\
\quad \mathit{opt}\ @_{sta}\ (\mathit{cfgf}\ @_{sta}\ (\mathit{bcf}\ @_{sta}\ x))\\
)
\end{array}
$$

Bien que cet exemple soit très simple, il montre déjà un des intérêts de SCOL : le niveau d'abstraction élevé. En effet, réaliser cette chaîne d'obfuscation directement dans LLVM demande une bonne connaissance du fonctionnement interne du compilateur, en particulier pour gérer l'ordre d'application des passes.

6.2.2 Sémantique d'évaluation de SCOL

L'application d'un script d'obfuscation se fait sur des paramètres passés en entrée. Ces paramètres peuvent être des programmes, des structures de données contenant des programmes, d'autres expressions autorisées par le langage (cf. 6.3).

L'application du script à une valeur d'entrée sera notée : $(\texttt{script}(x) = e)\, v$. v est la valeur passée en entrée au script d'obfuscation, il s'agira le plus souvent d'un programme ou d'une structure de données contenant des programmes (avec les extensions décrites dans la suite de ce chapitre).

Les règles d'évaluation sont de la forme $e \Rightarrow_w v$ qui se lit : « l'expression e s'évalue en la valeur v dans le monde w » Les règles de la sémantique de l'application d'un script d'obfuscation sont données ci-dessous.

$$\textsc{Script}\ \frac{[x \mapsto v]e \Rightarrow_{sta} v'}{(\texttt{script}(x) = e)\, v \Longrightarrow v'}$$

$$\textsc{ScriptError}\ \frac{[x \mapsto v]e \Rightarrow_{sta} \textsc{Error}}{(\texttt{script}(x) = e)\, v \Longrightarrow \textsc{Error}}$$

Pour s'exécuter, un script d'obfuscation doit être appliqué à un paramètre d'entrée. La sémantique de cette application consiste à évaluer le corps du script en remplaçant

la variable d'entrée par la valeur donnée en paramètre. La substitution utilisée ici est la substitution standard. Le résultat de l'application d'un programme ne peut être qu'une valeur ou une erreur. Intuitivement, si le corps s'évalue en une valeur, le script s'évaluera dans cette valeur.

Dans les cas d'application usuels, la valeur résultat sera (ou contiendra) un programme : le programme obfusqué par le script. Cependant, le langage ne limite pas quelles sont les valeurs qui peuvent être obtenues en résultat de l'application d'un script [2]. De la même manière, le langage ne limite pas, en théorie, quelles valeurs peuvent être utilisée en paramètre d'entrée. Cependant, ces valeurs doivent être manipulables par le gestionnaire de passes qui exécute le script, or dans notre implémentation actuelle, les fonctions (plus précisément les λ-abstraction de $\texttt{SCOL}_{dyn}$) n'ont pas d'existence dans le gestionnaire de passes, elles ne peuvent donc pas être utilisées en tant que paramètre d'entrée.

La sémantique de l'application est la sémantique usuelle de l'application App : le résultat de l'évaluation de l'application est l'évaluation du corps de l'abstraction ${e_w}'$ avec la variable x substituée par la valeur du paramètre de l'application v. La règle AppErrorI gère un cas d'erreur : si le corps de la fonction s'évalue en une erreur, alors l'application s'évalue en une erreur. Les règles AppErrorII et AppErrorIII indiquent que si un des membres de l'application s'évalue en une erreur, alors l'application s'évalue en une erreur.

$$\textsc{App}\ \frac{e_w \Rightarrow_w \lambda_{dyn} x.{e_{dyn}}' \qquad {e_w}'' \Rightarrow_w v \qquad [x \mapsto v]{e_{dyn}}' \Rightarrow_{dyn} v'}{e_w \mathbin{@_w} {e_w}'' \Rightarrow_w v'}$$

$$\textsc{AppErrorI}\ \frac{e_w \Rightarrow_w \lambda_{dyn} x.{e_{dyn}}' \qquad {e_w}'' \Rightarrow_w v \qquad [x \mapsto v]{e_{dyn}}' \Rightarrow_{dyn} \textsc{Error}}{e_w \mathbin{@_w} {e_w}'' \Rightarrow_w \textsc{Error}}$$

$$\textsc{AppErrorII}\ \frac{{e_w}'' \Rightarrow_w \textsc{Error}}{e_w \mathbin{@_w} {e_w}'' \Rightarrow_w \textsc{Error}} \qquad \textsc{AppErrorIII}\ \frac{e_w \Rightarrow_w \textsc{Error}}{e_w \mathbin{@_w} {e_w}'' \Rightarrow_w \textsc{Error}}$$

Le langage possède plusieurs règles sémantiques pour la gestion de la déclaration des passes et leur application :

2. Dans le chapitre suivant, nous verrons que le typage contraindra les résultats possibles.

$$\textsc{LetPass}\ \frac{[x \mapsto \texttt{pass}(\text{PASS})]e \Rightarrow_{sta} v}{\texttt{let\_pass}\ x\ \texttt{from}\ \text{PASS}\ \texttt{in}\ e \Rightarrow_{sta} v}$$

$$\textsc{RunPass}\ \frac{e_{dyn} \Rightarrow_{dyn} v_{dyn} \qquad \{\text{PASS}, p, cond\} \in \mathbb{P} \qquad cond(v_{dyn})}{\texttt{RunPass}(\text{PASS}, e_{dyn}) \Rightarrow_{dyn} p(v_{dyn})}$$

$$\textsc{AppPass}\ \frac{e_w \Rightarrow_w \texttt{pass}(\text{PASS}) \qquad e_w{}'' \Rightarrow_w v \qquad \{\text{PASS}, p, cond\} \in \mathbb{P} \qquad cond(v)}{e_w \mathbin{@}_w e_w{}'' \Rightarrow_w p(v)}$$

$$\textsc{PassError}\ \frac{\text{PASS} \notin dom(\mathbb{P})}{\texttt{RunPass}(\text{PASS}, e_{dyn}) \Rightarrow_{dyn} \textsc{Error}}$$

$$\textsc{PassErrorII}\ \frac{e_{dyn} \Rightarrow_{dyn} v_{dyn} \qquad \{\text{PASS}, p, cond\} \in \mathbb{P} \qquad \neg cond(v_{dyn})}{\texttt{RunPass}(\text{PASS}, e_{dyn}) \Rightarrow_{dyn} \textsc{Error}}$$

La règle LetPass exprime la sémantique de l'expression `let_pass` x `from` PASS `in` e. Cette expression définit une passe et évalue le terme e en remplaçant la variable x par la passe. Cette sémantique est donc similaire à celle d'un *let* dans un λ-calcul usuel, d'où la forme de cette expression.

Pour chaque passe définie dans le langage, il existe une fonction p dans l'ensemble des passes $\mathbb{P}$ qui représente l'implémentation de la passe. L'application des passes est donc similaire à une application usuelle, à la différence que le corps de la passe n'est pas connu dans le langage (à la différence des fonctions) et il se fait donc via une fonction externe : p. Cette application est exprimée par les règles RunPass et AppPass. Une passe ne peut être appliquée que si cette fonction p existe, c'est-à-dire si elle a été définie dans le gestionnaire de passes.

La règle PassError indique que l'application d'une passe qui n'a pas d'implémentation, c'est-à-dire qui n'appartient pas au domaine de $\mathbb{P}$, résulte en une erreur. $\mathbb{P}$ contient également les conditions de l'application de la passe, telles que décrites dans le chapitre 5. Cette condition permet de définir la sémantique de PassErrorII, qui exprime le fait qu'une passe ne peut pas être appliquée si les conditions d'application ne sont pas remplies.

La définition précise de ces conditions d'application est réalisée dans le système de

types (cf. chapitre 7). On ne définit pas formellement ici cette fonction de vérification de condition, dans la pratique on considèrera que si une fonction est bien typée, alors la condition d'application de la passe est respectée. Autrement dit, la charge de la vérification de la condition d'application est laissée au système de types.

Exemple

Reprenons l'exemple précédent :

$$
\begin{array}{l}
\texttt{script}(x) = (\\
\quad \texttt{let\_pass}\ \mathit{bcf}\ \texttt{from}\ \textsc{Obf.BCF}\ \texttt{in}\\
\quad \texttt{let\_pass}\ \mathit{cfgf}\ \texttt{from}\ \textsc{Obf.CFGF}\ \texttt{in}\\
\quad \texttt{let\_pass}\ \mathit{opt}\ \texttt{from}\ \textsc{Opt.Generic}\ \texttt{in}\\
\quad \mathit{opt}\ @_{sta}\ (\mathit{cfgf}\ @_{sta}\ (\mathit{bcf}\ @_{sta}\ x))\\
)
\end{array}
$$

Si on applique ce script d'obfuscation à un programme `prog`, la sémantique de cette application est la suivante :

$$
\begin{array}{l}
p_3(p_2(p_1(\texttt{prog})))\\
\qquad si\ c_1(\texttt{prog}) \wedge c_2(p_1(\texttt{prog})) \wedge c_3(p_2(p_1(\texttt{prog})))\\
\qquad avec \{(\textsc{Obf.BCF}, p_1, c_1), (\textsc{Obf.CFGF}, p_2, c_2), (\textsc{Opt.Generic}, p_3, c_3)\} \in \mathbb{P}
\end{array}
$$

Pour que ce script s'évalue en une valeur, il est nécessaire que pour chacune de ces passes, il existe la fonction d'application de la passe dans $\mathbb{P}$ et que les conditions d'applications de chacune des passes soient respectées. Si l'une des passes n'est pas contenue dans $\mathbb{P}$ ou au moins l'une des conditions d'application n'est pas respectée, le résultat de l'application sera une erreur (selon les règles sémantiques définies précédemment).

Cet exemple montre un des autres intérêts de SCOL, les passes ne sont appliquées que si leurs conditions d'application sont respectées. L'infrastructure de compilation LLVM ne propose pas de mécanismes pour gérer les contraintes d'applications des passes, ce sont les passes elles-mêmes qui doivent gérer ces contraintes. Avec LLVM, un utilisateur pourra difficilement savoir si une passe a été appliquée ou non et si elle a été appliquée efficacement, car il s'agit de la responsabilité individuelle de la passe qui peut utiliser des conventions de gestion des erreurs différentes des autres passes. Dans SCOL, cette problématique est gérée par le gestionnaire de passes directement, et permet d'assurer que

les passes ont été appliquées correctement et de connaitre les contraintes non respectées si les passes ne peuvent pas être appliquées.

6.2.3 Sémantique de traduction de SCOL

Maintenant que nous avons défini la syntaxe des deux langages composant SCOL, nous pouvons définir la sémantique de traduction du script d'obfuscation. La sémantique utilise un environnement E pour l'évaluation des termes. Cet environnement fait l'association entre une variable et l'expression dynamique à laquelle elle est liée. La traduction se fait en 2 étapes : une traduction depuis le langage statique vers le langage dynamique et une évaluation (toujours statique) dans le langage dynamique. Les règles dans le langage statique sont indiquées par $\vdash_{sta}$ et $\Downarrow_{sta}$. Les règles dans le langage dynamique sont indiquées par $\vdash_{dyn}$ et $\Downarrow_{dyn}$. Certaines règles sont valables dans les deux langages et sont indicées par w.

Une règle sémantique dans le langage statique est sous la forme $E \vdash_{sta} e_{sta} \Downarrow_{sta} e_{dyn}$ et une règle dans le langage dynamique sera de la forme $E \vdash_{dyn} e_{dyn} \Downarrow_{dyn} e_{dyn}$.

Cette évaluation en deux étapes permet de réaliser une partie de l'évaluation du programme statiquement, c'est-à-dire qu'une partie du script est évaluée à la compilation du script plutôt qu'à l'exécution de ce script. Les règles de traduction dans le monde dynamique sont nécessaires pour réaliser concrètement et statiquement les applications statiques : la règle StaticApp décrite page 100 évalue statiquement dans le monde dynamique l'application d'une abstraction. Plusieurs exemples dans ce chapitre montrent plus précisément l'intérêt du processus de compilation.

Commençons par les règles pour la traduction du terme principal :

$$\textsc{script}\ \frac{\{x \mapsto x\} \vdash_{sta} e \Downarrow_{sta} e_{dyn}}{\texttt{script}(x) = e \Downarrow \lambda_{dyn} x.e_{dyn}}$$

$$\textsc{scriptError}\ \frac{\{x \mapsto x\} \vdash_{sta} e \Downarrow_{sta} \textsc{Error}}{\texttt{script}(x) = e \Downarrow \textsc{Error}}$$

On traduit simplement le corps du script. Le résultat est soit une erreur, soit une fonction dans le langage dynamique. Dans le langage dynamique, nous ne distinguons plus le terme principal du programme des autres fonctions du langage, qui a une sémantique

similaire à une fonction (mais appliqué à des paramètres d'entrées, plutôt qu'à des termes du langage). D'un point de vue sémantique, dans le langage dynamique, il n'y a pas de différence entre le terme principal du script et les autres abstractions.

Les règles suivantes pour les variables et les abstractions sont proches des règles usuelles, les principales différences étant que nous utilisons un environnement qui ne contient pas que des valeurs (règles VAR et VARERROR) et que nous évaluons les expressions à l'intérieur de l'abstraction (règles LAMBDA et LAMBDAERROR) :

$$\textsc{Var}\ \frac{\{x \mapsto e_{dyn}\} \in E}{E \vdash_w x \Downarrow_w e_{dyn}} \qquad \textsc{VarError}\ \frac{x \notin dom(E)}{E \vdash_w x \Downarrow_w \textsc{Error}}$$

$$\textsc{Lambda}\ \frac{E \cup \{x \mapsto x\} \vdash_w e_w \Downarrow_w e_{dyn}}{E \vdash_w \lambda_w x.e_w \Downarrow_w \lambda_{dyn} x.e_{dyn}}$$

$$\textsc{LambdaError}\ \frac{E \cup \{x \mapsto x\} \vdash_w e_w \Downarrow_w \textsc{Error}}{E \vdash_w \lambda_w x.e_w \Downarrow_w \textsc{Error}}$$

Pour simplifier l'écriture de certaines règles de traduction, nous définissons des contextes de traductions. Ceux-ci permettent de regrouper les règles de traduction de plusieurs constructions qui sont traduites de façon similaire. Cela sera particulièrement utile lorsque nous ajouterons de nouvelles constructions dans la suite du chapitre. Les contextes sont les suivants :

$$S_{sta}[\_] ::= \_ \ @_{sta}\ e_{sta} \mid e_{sta}\ @_{sta}\ \_$$
$$S_{dyn}[\_] ::= \_ \ @_{dyn}\ e_{dyn} \mid e_{dyn}\ @_{dyn}\ \_ \mid \texttt{RunPass}(\textsc{pass}, \_)$$

Concernant les règles pour l'application, nous avons fait plusieurs choix. Nous avons décidé que l'application de fonction est réalisée par la règle STATICAPP lorsque la fonction et l'argument sont connus statiquement. Lorsqu'au moins l'un des deux termes est dynamique, l'application n'est pas effectuée, comme l'indique la règle CONTEXT, qui généralise la traduction des expressions à l'intérieur des termes définis dans les contextes. Il aurait été possible de faire l'application si la fonction est connue statiquement, quel que soit l'argument, ce qui correspondrait à toujours expanser les fonctions (remplacer l'appel de la fonction par son corps). Nous avons décidé de ne pas le faire pour éviter de dupliquer les corps de fonction. Si le terme de gauche de l'application est une valeur qui n'est pas

une fonction (c'est-à-dire une λ-abstraction ou une passe) alors l'application résulte en une erreur. Avec cette définition du langage, cela n'est pas possible, car les seules valeurs exprimables sont des fonctions, mais dans la suite du chapitre nous allons ajouter d'autres valeurs qui ne sont pas des fonctions, ce qui explique la présence de cette condition.

Notons que le contexte dans le langage statique S_{sta} est traduit dans le contexte équivalent dynamique S_{dyn}. Cela est possible, car chaque contexte statique à un équivalent dynamique [3].

Enfin, la règle ContextError extrait les erreurs du contexte, c'est-à-dire que si une expression d'un contexte se traduit en une erreur, le contexte se traduira en une erreur aussi. De plus, on ne peut réaliser une application que si l'expression à gauche est une fonction. Donc, si cette expression est une valeur qui n'est pas une fonction, alors l'application se traduira également en une erreur (règle AppError).

$$\textsc{StaticApp}\ \frac{E \vdash_w e_1 \Downarrow_w \lambda_{dyn} x.e_{dyn} \qquad E \vdash_w e_2 \Downarrow_w v \qquad E \cup \{x \to v\} \vdash_{dyn} e_{dyn} \Downarrow_{dyn} e'_{dyn}}{E \vdash_w e_1 \ @_w \ e_2 \Downarrow_w e'_{dyn}}$$

$$\textsc{Context}\ \frac{E \vdash_w e \Downarrow_w e' \qquad e' \notin \mathbb{V}}{E \vdash_w S_w[e] \Downarrow_w S_{dyn}[e']}$$

$$\textsc{ContextError}\ \frac{E \vdash_w e \Downarrow_w \textsc{Error}}{E \vdash_w S_w[e] \Downarrow_w \textsc{Error}}$$

$$\textsc{AppError}\ \frac{E \vdash_w e_1 \Downarrow_w v \qquad E \vdash_w e_2 \Downarrow_w {e_{dyn}}' \qquad v \notin \mathit{function}}{E \vdash_w e_1 \ @_w \ e_2 \Downarrow_w \textsc{Error}}$$

Pour les passes, nous avons fait le choix de toujours les appliquer statiquement lorsqu'elles sont connues statiquement, ce qui est exprimé par la règle PassApp. L'expression `let_pass` x `from` PASS `in` e possède une sémantique, définie par la règle LetPass, ressemblante à celle d'un *let* classique. L'expression à l'intérieur est traduite en ajoutant l'association de la variable avec la passe définie par le `let_pass` dans l'environnement.

3. La réciproque est fausse par contre.

$$\textsc{PassApp}\ \frac{E \vdash_w e_1 \Downarrow_w \texttt{pass}(\textsc{pass}) \qquad E \vdash_w e_2 \Downarrow_w e_{dyn}}{E \vdash_w e_1 \mathbin{@_w} e_2 \Downarrow_w \texttt{RunPass}(\textsc{pass}, e_{dyn})}$$

$$\textsc{LetPass}\ \frac{E \cup \{x \mapsto \texttt{pass}(\textsc{pass})\} \vdash_{sta} e \Downarrow_{sta} e_{dyn}}{E \vdash_{sta} \texttt{let\_pass}\ x\ \texttt{from}\ \textsc{pass}\ \texttt{in}\ e \Downarrow_{sta} e_{dyn}}$$

Exemple de traduction

Reprenons notre script SCOL d'exemple utilisé précédemment :

```
script(x) = (
  let_pass bcf from Obf.BCF in
  let_pass cfgf from Obf.CFGF in
  let_pass opt from Opt.Generic in
  opt @sta (cfgf @sta (bcf @sta x))
)
```

En appliquant successivement les règles sémantiques LetPass, PassApp et Var définies précédemment on obtient la sémantique suivante pour le programme :

```
λdyn x.(
  RunPass(Opt.Generic,
    RunPass(Obf.CFGF,
      RunPass(Obf.BCF, x)))
)
```

Si on examine le résultat de l'évaluation du script on observe que l'on applique successivement sur le programme d'entrée x les transformations Obf.BCF, puis, Obf.CFGF et enfin Opt.Generic ce qui est le comportement que l'on recherchait.

6.3 Extensions au langage

Le langage SCOL que nous avons présenté précédemment permet d'écrire des scripts d'obfuscation. Mais ces scripts d'obfuscation sont limités et sont donc peu utiles. On ne peut écrire que des scripts qui appliquent une séquence de passes dans un ordre fixé sans

aucun paramètre.

Pour permettre d'écrire des scripts d'obfuscation plus intéressants, nous allons enrichir SCOL d'extensions qui sont des extensions traditionnelles au λ-calcul [169]. En particulier, nous allons ajouter les entiers, les booléens, les opérateurs binaires et arithmétiques courants, les structures conditionnelles, les tuples, les listes et les fonctions récursives. Nous ne présenterons pas formellement ici la sémantique d'évaluation de ces extensions, elle est standard et est détaillée en annexe B.

Chacune des extensions présentées ici est indépendante et peut être utilisée simultanément avec les autres ou non.

6.3.1 Booléens et structures conditionnelles

Pour permettre du dynamisme dans les scripts d'obfuscation, nous allons ajouter des structures conditionnelles au langage. En plus des constructions pour les structures conditionnelles nous avons besoin de booléens pour utiliser ces structures.

On ajoute donc d'abord deux nouvelles expressions : `true` (le booléen vrai) et `false` (le booléen faux). Elles deviennent des valeurs du langage. Nous ajoutons également les opérateurs binaires logique `and` (`ET` logique), `or` (`OU` logique), ainsi que l'opérateur unaire `not` et l'égalité (=). Enfin, nous pouvons ajouter la structure conditionnelle `if` ... `then` ... `else` ....

Cela donne la syntaxe suivante pour SCOL :

$$
\begin{array}{lll}
\ldots & & \\
e_{sta} & ::= \ldots \mid \mathit{constant}_{sta} \mid \texttt{if}_{sta}\ e_{sta}\ \texttt{then}\ e_{sta}\ \texttt{else}\ e_{sta} \mid \mathit{unop}_{sta}\ e_{sta} \mid e_{sta}\ \mathit{binop}_{sta}\ e_{sta} \\
\mathit{constant}_{w} & ::= \texttt{true}_{w} \mid \texttt{false}_{w} & \\
\mathit{unop}_{w} & ::= \texttt{not}_{w} & \\
\mathit{binop}_{w} & ::= \texttt{and}_{w} \mid \texttt{or}_{w} \mid =_{w} & \\
e_{dyn} & ::= \ldots \mid \texttt{if}_{dyn}\ e_{dyn}\ \texttt{then}\ e_{dyn}\ \texttt{else}\ e_{dyn} \mid \mathit{unop}_{dyn}\ e_{dyn} \mid e_{dyn}\ \mathit{binop}_{dyn}\ e_{dyn} \\
v & ::= \ldots \mid \mathit{constant}_{dyn} & \\
S_{w}[\_] & ::= \ldots \mid \texttt{if}_{w}\ \_\ \texttt{then}\ e_{w}\ \texttt{else}\ e_{w} \mid \texttt{if}_{w}\ e_{w}\ \texttt{then}\ \_\ \texttt{else}\ e_{w} \mid & \\
 & \quad \texttt{if}_{w}\ e_{w}\ \texttt{then}\ e_{w}\ \texttt{else}\ \_ &
\end{array}
$$

La règle pour la construction `not` est la règle logique usuelle, ses règles sémantiques sont les suivantes :

$$\textsc{NotTrue}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{true}_{dyn}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \texttt{false}_{dyn}} \qquad \textsc{NotFalse}\ \frac{E \vdash e_w \Downarrow_w \texttt{false}_{dyn}}{E \vdash \texttt{not}_w\, e_w \Downarrow_w \texttt{true}_{dyn}}$$

$$\textsc{DynamicNot}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} \qquad e_{dyn} \notin \mathbb{V}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \texttt{not}_{dyn}\, e_{dyn}}$$

$$\textsc{NotError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \textsc{Error}}$$

De même pour les opérateurs binaires, on applique les règles logiques usuelles :

$$\textsc{ReducBinOp}\ \frac{\begin{array}{c} E \vdash_w e_w \Downarrow e_{dyn} \\ E \vdash e_w{}' \Downarrow e_{dyn}{}' \qquad op \in binop_w \qquad \texttt{Red}(e_{dyn}\, op\, e_{dyn}{}') = e_{dyn}{}'' \end{array}}{E \vdash_w e_w\, op_w\, e_w{}' \Downarrow_w e_{dyn}{}''}$$

Lorsque l'un des deux membres d'un opérateur binaire est une valeur, il est possible de réduire statiquement l'expression avec les règles logiques classiques. C'est l'objectif de la règle ReducBinOp, qui utilise la fonction `Red` définie de la manière suivante :

$$\begin{array}{lcll}
\texttt{Red}(e_{dyn}\, \texttt{or}_{dyn}\, \texttt{true}_{dyn}) & = & \texttt{Red}(\texttt{true}_{dyn}\, \texttt{or}_{dyn}\, e_{dyn}) & = \texttt{true}_{dyn} \\
\texttt{Red}(e_{dyn}\, \texttt{or}_{dyn}\, \texttt{false}_{dyn}) & = & \texttt{Red}(\texttt{false}_{dyn}\, \texttt{or}_{dyn}\, e_{dyn}) & = e_{dyn} \\
\texttt{Red}(e_{dyn}\, \texttt{and}_{dyn}\, \texttt{true}_{dyn}) & = & \texttt{Red}(\texttt{true}_{dyn}\, \texttt{and}_{dyn}\, e_{dyn}) & = e_{dyn} \\
\texttt{Red}(e_{dyn}\, \texttt{and}_{dyn}\, \texttt{false}_{dyn}) & = & \texttt{Red}(\texttt{false}_{dyn}\, \texttt{and}_{dyn}\, e_{dyn}) & = \texttt{false}_{dyn} \\
\texttt{Red}(e_{dyn} =_{dyn} \texttt{true}_{dyn}) & = & \texttt{Red}(\texttt{true}_{dyn} =_{dyn} e_{dyn}) & = e_{dyn} \\
\texttt{Red}(e_{dyn} =_{dyn} \texttt{false}_{dyn}) & = & \texttt{Red}(\texttt{false}_{dyn} =_{dyn} e_{dyn}) & = \texttt{not}_{dyn} e_{dyn} \\
\texttt{Red}(e_{dyn}\, op_{dyn}\, e_{dyn}{}') & = & \textsc{Error} & \texttt{if}(e_{dyn} \in \mathbb{V} \setminus \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}) \\
 & & & \vee(e_{dyn}{}' \in \mathbb{V} \setminus \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\})
\end{array}$$

Enfin voici les règles pour la structure conditionnelle :

$$\textsc{IfTrue}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{true}_{dyn} \qquad E \vdash e_w{}' \Downarrow e_{dyn}}{E \vdash_w \texttt{if}\, e_w\, \texttt{then}\, e_w{}'\, \texttt{else}\, e_w{}'' \Downarrow_w e_{dyn}}$$

$$\textsc{IfFalse}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{false}_{dyn} \qquad E \vdash_w e_w{}'' \Downarrow_w e_{dyn}}{E \vdash_w \texttt{if}\, e_w\, \texttt{then}\, e_w{}'\, \texttt{else}\, e_w{}'' \Downarrow_w e_{dyn}}$$

$$\textsc{IfError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{E \vdash_w \texttt{if}_w\, e_w\, \texttt{then}\, e_w{}'\, \texttt{else}\, e_w{}'' \Downarrow \textsc{Error}}$$

Le premier argument de cette construction est la condition de la structure. Si cette condition est vraie le résultat de la structure est l'expression correspondante au `then` (règle IfTrue) alors que si la condition est fausse le résultat sera l'expression du `else` (règle IfFalse). Si la condition de la structure est une valeur, mais n'est pas un booléen, alors la règle IfError nous indique que la structure conditionnelle se traduit en une erreur. Dans certains cas, l'expression de la condition ne pourra pas être évaluée comme un booléen dans la compilation du script SCOL. Dans ces cas, on utilisera la règle Context. Cela arrivera notamment lorsque cette condition dépend du résultat d'une analyse, la valeur sera dans ce cas un `RunPass`(PASS, x).

Exemple

Prenons un script d'obfuscation qui effectue une analyse sur un programme, puis selon le résultat de cette analyse, il applique ou non une passe de transformation sur le programme.

Un tel script d'obfuscation pourra s'écrire en SCOL comme suit :

```
script(x) = (
  let_pass analysis from ANALYSIS.ANA in
  let_pass pass from OBF.PASS in
  if_sta (analysis @_sta x) then (pass @_sta x) else x)
```

La traduction dans $\texttt{SCOL}_{dyn}$ de ce script, en appliquant les règles précédentes sera :

$\lambda_{dyn} x.($
$\quad \texttt{if}_{dyn}\,(\texttt{RunPass}(\textsc{Analysis.ana}, x))\,\texttt{then}\,(\texttt{RunPass}(\textsc{Obf.pass}, x))\,\texttt{else}\,x)$

La condition du `if` est le résultat d'une passe, il s'agit donc d'un terme dynamique, car on ne connaitra le résultat qu'à l'application de la passe. Le script traduit contiendra donc la structure conditionnelle.

L'application de passes selon le résultat de l'application d'autres passes n'est pas une mécanique prévue dans les compilateurs traditionnels [4], et demande donc une bonne connaissance du fonctionnement interne du compilateur ou une modification de l'implémentation des passes. Alors que comme cet exemple le montre, cela peut être réalisé plus simplement avec SCOL.

6.3.2 Nombres entiers

Pour offrir plus de possibilités dans la description de chaîne d'obfuscation, on intègre également les nombres entiers à notre langage. Les calculs d'entiers peuvent par exemple être utilisés en résultat d'analyse pour déterminer si une passe doit être appliquée ou non (si le résultat dépasse un seuil ou non).

Pour cela, nous ajoutons en plus des nombres entiers, les opérations arithmétiques classiques (addition, soustraction, division entière, multiplication et modulo), ainsi que les comparaisons ($<$, $>$, $\geq$, $\leq$, $=$ et $\neq$).

La syntaxe du langage devient donc :

$$
\begin{array}{ll}
\ldots & \\
\mathit{constant}_w & ::= \ldots \mid \mathit{int}_w \\
\mathit{binop}_w & ::= \ldots \mid <_w \mid >_w \mid =_w \mid \leq_w \mid \geq_w \mid \neq_w \mid +_w \mid -_w \mid /_w \mid *_w \mid \texttt{mod}_w
\end{array}
$$

Les règles sémantiques correspondent bien sûr aux règles mathématiques usuelles sur les nombres entiers et sont appliquées via la règle ReducBinOp définie précédemment. Nous utilisons à nouveau la fonction `Red` définie précédemment. Nous ne donnerons pas ici la définition précise de cette extension aux entiers, car elle est complexe à écrire et peu intéressante.

4. En dehors de la chaîne d'optimisation fixée dans le compilateur.

Intuitivement, cette fonction réduit les calculs arithmétiques à base d'entier en calculant les valeurs qui ne contiennent pas de variables (potentiellement en réalisant une réécriture du calcul selon les règles mathématiques usuelles de ces opérateurs). Par exemple, sans écrire formellement l'ensemble des règles, on peut retirer les réductions imbriquées : $\texttt{Red}(e+\texttt{Red}(e')) = \texttt{Red}(e+e')$. Ensuite, comme l'addition est symétrique, on a également : $\texttt{Red}(e+e') = \texttt{Red}(e'+e)$. Enfin, si les deux opérandes d'un opérateur sont des entiers, on peut simplement calculer le résultat du calcul, par exemple : $\texttt{Red}(1+2) = 3$. Cela permet de réduire l'expression $1 + x + y * 3 + 4$ en $5 + x + y * 3$. La règle d'erreur est également un peu plus complexe à cause de ces règles de calculs supplémentaires.

6.3.3 Listes

Les listes sont des structures de données qui permettent de stocker une séquence d'éléments de même type. Une liste est construite à partir d'une liste vide ($[]$) à laquelle des éléments peuvent être ajoutés avec l'opérateur ::. Par exemple, on peut décrire la liste qui contient les éléments v, v' et v'' par $v'' :: v' :: v :: []$. Pour simplifier les notations, nous noterons parfois une liste de n éléments sous la forme $v_1 :: ... :: v_n$.

Pour manipuler les listes, nous utiliserons les constructions `hd` (*head* donnant le premier élément de la liste) et `tl` (*tail* la liste privée de son premier élément). Enfin, nous ajoutons un cas d'erreur, appliquer les opérations `hd` et `tl` sur une liste vide génère une erreur. La syntaxe de SCOL avec les listes est la suivante :

$$
\begin{array}{ll}
... & \\
e_{sta} & ::= ... \mid e_{sta} ::_{sta} e_{sta} \mid \texttt{hd}_{sta}\, e_{sta} \mid \texttt{tl}_{sta}\, e_{sta} \\
constant_w & ::= ... \mid []_w \\
e_{dyn} & ::= ... \mid \texttt{hd}_{dyn}\, e_{dyn} \mid \texttt{tl}_{dyn}\, e_{dyn} \\
v & ::= ... \mid e_{dyn} ::_{dyn} e_{dyn} \\
S_w[\_] & ::= ... \mid \_ ::_w e_w \mid e_w ::_w \_ \mid \texttt{hd}_w\_ \mid \texttt{tl}_w\_
\end{array}
$$

Et la sémantique de ces nouvelles constructions sera la suivante :

$$\textsc{StaticHd}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} ::_{dyn} e_{dyn}' \qquad e_{dyn} \neq \textsc{Error} \qquad e_{dyn}' \neq \textsc{Error}}{E \vdash_w \texttt{hd}_w\, e_w \Downarrow_w e_{dyn}}$$

$$\textsc{StaticTl}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} ::_{dyn} e_{dyn}' \qquad e_{dyn} \neq \textsc{Error} \qquad e_{dyn}' \neq \textsc{Error}}{E \vdash_w \texttt{tl}_w\, e_w \Downarrow_w e_{dyn}'}$$

$$\textsc{HdError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq e_{dyn} ::_{dyn} e_{dyn}'}{E \vdash_w \texttt{hd}_w\, e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{TlError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq e_{dyn} ::_{dyn} e_{dyn}'}{E \vdash_w \texttt{tl}_w\, e_w \Downarrow_w \textsc{Error}}$$

Si la construction de la liste est statique, c'est à dire si l'expression a la forme $e :: e'$, alors on peut appliquer statiquement les opérateurs `hd` et `tl` avec les règles StaticHd et StaticTl, sinon ces opérations devront être appliquées dynamiquement. La règle Context est alors utilisée pour traduire l'expression. On ne peut appliquer les opérateurs `hd` et `tl` que sur des listes, donc si l'opérande est une valeur, mais n'est pas une liste, l'opération donnera une erreur avec les règles HdError et TlError.

6.3.4 Tuples

Comme les listes, les tuples sont des structures de données contenant une séquence d'éléments. Mais contrairement aux listes, les tuples ont une taille fixée (il n'y a pas l'opérateur :: permettant d'étendre la liste) et les éléments peuvent être de types différents.

Le tuple qui contient les expressions de e_1 à e_n est noté $(e_1, ..., e_n)$. Pour manipuler les tuples, nous utilisons l'opération de projection. Elle est notée $\pi^i\, e$ et extrait le i^{e} élément du tuple (règle StaticTupleProj lorsqu'elle est faite statiquement, et Context sinon). Enfin, si l'on projette un tuple sur une dimension qui n'existe pas par rapport à la taille du tuple (règle TupleError), ou si on applique une projection sur une valeur qui n'est pas un tuple(règle TupleErrorII), on obtiendra une erreur.

La syntaxe de SCOL avec les tuples est la suivante :

$$
\begin{array}{lll}
 & ... & \\
e_{sta} & ::= ... \mid (e_{sta}, ..., e_{sta})_{sta} \mid \pi^i_{sta}\, e_{sta} \\
e_{dyn} & ::= ... \mid \pi^i_{dyn}\, e_{dyn} \\
v & ::= ... \mid (e_{dyn}, ..., e_{dyn})_{dyn} \\
S_w[\_] & ::= ... \mid (e_w, ..., \_, ..., e_w)_w \mid \pi^i_w\, \_
\end{array}
$$

Et la sémantique correspondante aux règles énoncées précédemment est :

$$\textsc{StaticTupleProj}\ \frac{E \vdash_w e_w \Downarrow_w (e^1_{dyn}, \ldots, e^i_{dyn}, \ldots, e^n_{dyn})_{dyn}}{E \vdash_w \pi^i_w\, e_w \Downarrow_w e^i_{dyn}}$$

$$\textsc{TupleError}\ \frac{E \vdash_w e_w \Downarrow_w (e^1_{dyn}, \ldots, e^n_{dyn})_{dyn} \qquad i > n \vee i < 1}{E \vdash_w \pi^i_w e \Downarrow_w \textsc{Error}}$$

$$\textsc{TupleErrorII}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq (e^1_{dyn}, \ldots, e^n_{dyn})_{dyn}}{E \vdash_w \pi^i_w e_w \Downarrow_w \textsc{Error}}$$

6.3.5 Récursion

Pour rendre notre langage de script d'obfuscation Turing complet, nous ajoutons finalement la possibilité de réaliser des récursions. La récursion permet, en particulier lorsqu'elle est combinée aux autres extensions proposées, d'exécuter des passes en boucle jusqu'à atteindre un certain critère ou encore d'offrir plus de possibilités dans la manipulation de listes.

Pour cela nous ajoutons la fonction point fixe :

$$
\begin{array}{ll}
 & ... \\
e_{sta} & ::= ... \mid \mu_{sta} f.\lambda x.e_{sta} \\
function & ::= ... \mid \mu_{dyn} f.\lambda x.e_{dyn}
\end{array}
$$

Lors de l'application d'une fonction récursive à un argument statique la variable f de la fonction point fixe est substituée par la fonction elle-même.

La sémantique de cette construction est :

$$\textsc{StaticRec}\ \frac{E \vdash_{sta} e_{sta} \Downarrow_{sta} \mu_{dyn} f.\lambda x.e_{dyn} \qquad E \vdash_{sta} e_{sta}' \Downarrow_{sta} v \qquad E \cup \{f \mapsto \mu_{dyn} f.\lambda x.e_{dyn}, x \mapsto v\} \vdash_{dyn} e_{dyn} \Downarrow e_{dyn}'}{E \vdash_{sta} e_{sta} @_{sta} e_{sta}' \Downarrow_{sta} e_{dyn}'}$$

$$\textsc{DynamicRec}\ \frac{E \vdash_{dyn} e_{dyn} \Downarrow_{dyn} \mu_{dyn} f.\lambda x.e_{dyn}' \qquad E \vdash_{dyn} e_{dyn}'' \Downarrow_{dyn} e_{dyn}'''}{E \vdash_{dyn} e_{dyn} @_{dyn} e_{dyn}'' \Downarrow_{dyn} \mu_{dyn} f.\lambda x.e_{dyn}' @_{dyn} e_{dyn}'''}$$

$$\textsc{Mu}\ \frac{E \cup \{f \mapsto f, x \mapsto x\} \vdash_{w} e_{w} \Downarrow_{w} e_{dyn}}{E \vdash_{w} \mu_{w} f.\lambda x.e_{w} \Downarrow_{w} \mu_{dyn} f.\lambda x.e_{dyn}}$$

Pour la récursion, il y a des différences majeures entre les règles de traduction dans les mondes *sta* et *dyn* contrairement aux règles définies précédemment. En effet, l'application *sta* est similaire à l'application de λ, c'est-à-dire que si la fonction est appliquée sur une valeur, on réalise la substitution de la définition de la fonction et de la valeur à l'intérieur du corps de la fonction récursive (règle StaticRec). Par contre, pour l'application *dyn*, on ne substitue jamais l'argument dans le corps de la fonction (règle DynamicRec) pour éviter les récursions infinies. Autrement dit, on n'expanse jamais les fonctions récursives dans le monde *dyn*. Enfin, comme pour les abstractions, on traduit à l'intérieur de la fonction récursive avec la règle Mu. Ces règles permettent à la fois de faire certaines réductions statiquement, tout en assurant que la traduction soit finie (c'est-à-dire qu'elle ne génère pas une réduction infinie). Autrement dit, on ne « déroule » la récursion qu'une seule fois au maximum. Il s'agit d'un choix que nous avons fait pour notre langage pour un compromis entre la simplicité des règles de traduction et les possibilités de réductions statiques. Il aurait potentiellement été possible éventuellement de faire cette traduction en une seule étape ou d'avoir plus d'étapes dans la traduction. Ces méthodes auraient pu conserver la même sémantique d'exécution du script avec une sémantique de traduction différente, et donc même si le résultat final de l'application du script sera identique, le moment (à la compilation ou l'exécution) où les erreurs sont détectées et certains calculs effectués seront différents. On pourrait donc dans le futur implémenter ces sémantiques de traduction si elles apportent un intérêt pour certains scripts d'obfuscation.

Exemple

Nous pouvons utiliser la récursion pour définir dans SCOL la fonction *map*. Cette fonction applique une fonction, reçue en paramètre, à chaque élément d'une liste. Ses paramètres sont la fonction à appliquer et la liste sur laquelle la fonction doit être appliquée.

En SCOL, la fonction *map*, sera définie de la façon suivante :

$$\mu_{sta} map.\lambda f.\lambda_{sta} y.\texttt{if}_{sta}\ y = []\ \texttt{then}\ []\ \texttt{else}\ ((f\ @\ (\texttt{hd}_{sta}\ y))\ ::\ (map\ @_{sta}\ f\ @_{sta}\ (\texttt{tl}_{sta}\ y)))$$

On peut écrire un script utilisant cette fonction. Par exemple, un script qui prend en entrée une liste de programmes et qui applique une passe d'obfuscation à tous les programmes de la liste. Ce script pourra s'écrire de la façon suivante :

$$\begin{array}{l}\texttt{script}(x) = (\\ \quad \texttt{let\_pass}\ pass\ \texttt{from}\ \textsc{Obf.pass}\ \texttt{in}\\ \quad \mu_{sta} map.\lambda_{sta} f.\lambda_{sta} y.\texttt{if}_{sta}\ y = []_{sta}\ \texttt{then}\ []_{sta}\ \texttt{else}\ ((f\ @_{sta}\ (\texttt{hd}_{sta}\ y))\ ::_{sta}\ (map\ @_{sta}\ f\ @_{sta}\ (\texttt{tl}_{sta}\ y)))\\ \qquad @_{sta}\ pass\ @_{sta}\ x\\ )\end{array}$$

En utilisant les règles de traduction définies dans ce chapitre, on obtiendra la traduction suivante pour ce script :

$$\begin{array}{l}\lambda_{dyn} x.(\\ \quad \mu_{sta} map.\lambda_{sta} y.\\ \qquad \texttt{if}_{sta}\ y = []_{sta}\ \texttt{then}\ []_{sta}\ \texttt{else}\ ((\texttt{pass}(\textsc{pass})\ @_{dyn}\ (\texttt{hd}_{sta} y))\ ::_{sta}\\ \qquad\quad (map\ @_{dyn}\ \texttt{pass}(\textsc{pass})\ @_{dyn}\ (\texttt{tl}_{sta} y)))\\ \qquad\quad @_{dyn}\ x\\ )\end{array}$$

On remarque dans cet exemple que l'évaluation statique a permis de spécialiser la fonction récursive *map* à l'application de la passe PASS au site d'application.

L'utilisation de cette fonction, ainsi que d' autres fonctions que l'on peut implémenter (filter, reduce ...), dans des scripts d'obfuscation SCOL simplifie l'écriture et la compréhension de ces scripts. En effet, SCOL permet d'exprimer simplement des concepts (ici les fonctions et la récursion) qui ne sont généralement pas présents dans les chaînes de compilation. SCOL offre donc plus d'expressivité que les compilateurs traditionnels dans

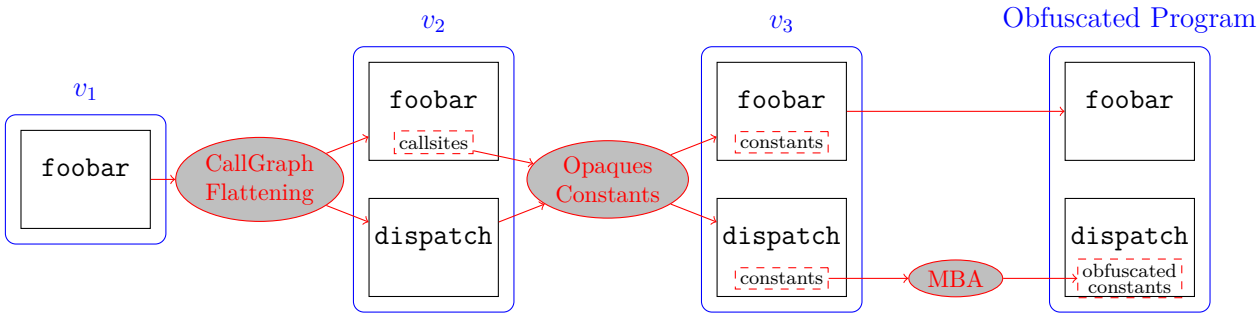

FIGURE 6.1 – Un exemple de chaîne d'obfuscation

la conception de chaînes de compilation.

6.4 Exemple de programme SCOL

En 5.4, nous avons présenté un scénario de compilation (fig. 6.1). Nous allons maintenant décrire ce processus à l'aide de SCOL.

6.4.1 Représentation du programme cible et des passes

Le programme cible dans notre exemple est composé de 3 fonctions : `foo`, `bar` et `foobar`. Ce programme sera l'entrée de notre script d'obfuscation, et nous allons le représenter par un tuple de 3 éléments dans lequel chaque élément correspondra à une fonction du programme. En sortie du script de compilation, nous aurons un tuple qui contiendra l'ensemble des fonctions du programme à la fin du processus d'obfuscation.

La première étape de l'écriture du script de compilation va être de déclarer l'ensemble des passes que nous allons utiliser dans notre processus :

— La passe de *CallGraph Flattening*, implémentée en OBF.CGF. L'entrée de cette passe est une fonction et la sortie sera un triplet contenant deux fonctions (la fonction protégée et la fonction *dispatcher*) et une liste d'entiers (la liste des entiers qui relie les appels de fonctions au *dispatcher*)(cf. 2.5.2).

— La passe de création des constantes opaques qui prend en entrée une paire contenant une fonction et une liste d'entier et retournant une fonction protégée (cf. 2.5.3). Elle sera implémentée en OBF.OPQCST.

— La passe de *Mixed-Boolean Arithmetic*, prenant en entrée une fonction et donnant

en sortie une nouvelle fonction. Elle est implémentée en Obf.MBA.

6.4.2 Script d'obfuscation

Nous commençons par écrire la déclaration des 3 passes de notre exemple :

```
script(x) = (
  let_pass cgf from Obf.CGF in
  let_pass opqcst from Obf.OpqCst in
  let_pass mba from Obf.MBA in
  ...
)
```

Ensuite, on extrait la fonction `foobar` du tuple d'entrée (on considère qu'il s'agit de la 3e fonction du tuple) :

```
...
let foobar = π³ x in
...
)
```

On applique ensuite les passes dans la séquence de passes décrite précédemment :

```
...
let result = cgf @ foobar in
let foobar2 = π¹ result in
let dispatch = π² result in
let const = π³ result in
let foobar3 = opqcst @ foobar2 @ const in
let dispatch2 = mba @ (opqcst @ dispatch @ const) in
...
)
```

On définit ici beaucoup de variables intermédiaires, il pourrait y en avoir moins et on pourrait faire appliquer les passes suivantes directement sur les résultats sans les stocker dans des variables intermédiaires.

Il ne reste ensuite plus qu'à former la sortie du script d'obfuscation et on obtient le script d'obfuscation complet suivant :

```
script(x) = (
  let_pass cgf from Obf.CGF in
  let_pass opqcst from Obf.OpqCst in
  let_pass mba from Obf.MBA in
  let foobar = π³ x in
  let result = cgf @ foobar in
  let foobar2 = π¹ result in
  let dispatch = π² result in
  let const = π³ result in
  let foobar3 = opqcst @ foobar2 @ const in
  let dispatch2 = mba @ (opqcst @ dispatch @ const) in
    (π¹ x, π² x, foobar3, dispatch2)
)
```

6.4.3 Traduction du script

En appliquant l'ensemble des règles d'évaluation présentées dans ce chapitre sur ce script d'obfuscation, on calcule la sémantique suivante :

```
λ_dyn x.(
  (π¹ x, π² x,
  RunPass(Obf.OpqCst,
    (π¹ RunPass(Obf.CGF, π³ x), (π³ RunPass(Obf.CGF, π³ x))),
  RunPass(Obf.MBA,
    RunPass(Obf.OpqCst,
      (π² RunPass(Obf.CGF, π³ x), (π³ RunPass(Obf.CGF, π³ x)))
))
```

L'évaluation de notre script SCOL exprime bien le comportement voulu pour le processus de compilation et d'obfuscation souhaité.

On remarque que du point de vue de notre langage, les applications des passes sont dupliquées, mais cela pourra être évité en gérant correctement le cache dans le gestionnaire de passes de LLVM (cf. 8).

6.4.4 Intérêts de l'utilisation de SCOL dans cet exemple

Cet exemple montre plusieurs intérêts de SCOL. D'abord, comme pour les exemples précédents de ce chapitre, l'utilisation des abstractions de haut niveau de SCOL permet d'écrire des scripts d'obfuscation sans connaissance avancée du compilateur. De plus, l'utilisation de SCOL permet aussi de choisir plus finement l'ordre et le lieu d'application des passes. En effet, LLVM ne contient pas de mécanismes permettant d'appliquer une passe à une fonction particulière : pour réaliser cela en LLVM, il faut modifier l'implémentation de la passe, et la rendre responsable des lieux d'applications. [5]. SCOL permet d'écrire ce script d'obfuscation sans connaissance du fonctionnement du compilateur et sans avoir à modifier les passes d'obfuscation spécifiquement cette chaîne d'obfuscation.

6.5 Conclusion

Dans ce chapitre, nous avons commencé à présenter notre langage dédié à la création de scripts d'obfuscation SCOL. Il s'agit d'un langage fonctionnel offrant des constructions particulières dédiées à la description de chaînes de compilation contenant des transformations d'obfuscation. Notamment, notre langage permet de gérer les concepts de passe, de programme et d'application de passes sur des programmes qui sont les concepts centraux de la description des chaînes de compilation. Notre langage offre plus d'expressivité que les méthodes et outils utilisés en pratique et permet d'avoir une description formelle des chaînes de compilation.

Nous avons présenté la syntaxe de notre langage, d'abord dans une version minimale, puis nous avons présenté les différentes extensions que nous avons ajoutées au langage pour le rendre Turing complet. En particulier, nous avons intégré les entiers, les booléens, les structures conditionnelles, les listes, les tuples et les fonctions récursives. Nous avons présenté les différentes sémantiques de SCOL. Dans un premier temps, nous avons présenté la sémantique d'évaluation de SCOL, c'est-à-dire la sémantique donnant le résultat de l'application d'un script SCOL sur un programme. Puis nous avons présenté la sémantique de traduction de SCOL, qui exprime la manière dont les premières étapes de la compilation de SCOL vers le C++ intégré au gestionnaire de passes de LLVM [6].

Dans les chapitres suivants, nous présenterons d'autres aspects de SCOL, de sa compilation et de son exécution. Dans le chapitre 7, nous présentons le système de types de

5. Le gestionnaire de passe de LLVM appelle les passes sur toute les fonctions du programme (cf. 3.3).
6. La suite sera décrite dans le chapitre 8.

SCOL. Ce système de types permet d'assurer la correction des scripts SCOL, en particulier que les passes sont appliquées correctement (tel que décrit dans le chapitre 5). L'utilisation du système de types implique quelques légères différences dans la syntaxe et la sémantique de SCOL, que nous avons omises dans ce chapitre pour des raisons pédagogiques. Elles seront décrites rapidement dans le chapitre suivant, et la description complète du langage est fournie en Annexes A et B. Dans le chapitre 8, nous présenterons la sémantique de traduction vers le C++ utilisé par le gestionnaire de passes, et le fonctionnement de notre gestionnaire de passes dans l'exécution de la chaîne de compilation décrites par le script SCOL.

Chapitre 7

Système de types de SCOL

7.1 Introduction

Dans le chapitre précédent, nous avons présenté notre langage SCOL dédié à la création de scripts d'obfuscation et nous en avons décrit la syntaxe et la sémantique. Cela permet de décrire des scripts d'obfuscation, mais ne permet pas de gérer les conditions d'application des passes ni d'obtenir les informations sur les propriétés des programmes après l'application des passes (tel que nous l'avons décrit dans le chapitre 5).

Dans ce chapitre, nous allons présenter le système de types du langage SCOL. Celui-ci a plusieurs objectifs : vérifier que les conditions d'application des passes sont correctes avant leur application, effectuer des calculs sur les propriétés des programmes et obtenir les propriétés des programmes résultants de l'exécution du script d'obfuscation SCOL. Pour cela, nous présenterons d'abord les types des constructions spécifiques à SCOL : les passes et les programmes. Puis nous présenterons le système de types de la version minimale de SCOL que nous avons présentée dans le chapitre 6. Ensuite, nous présenterons le typage des différentes extensions que nous avons ajoutées à SCOL, pour obtenir le système de types complet. Enfin, nous présenterons une application du système sur un exemple.

7.2 Système de types dans notre langage

7.2.1 Introduction

Nous allons maintenant présenter le système de types de SCOL. Nous allons commencer par une présentation des types des deux éléments principaux que nous avons ajoutés au λ-calcul pour l'utiliser comme langage de description et d'exécution de chaînes de compilation et d'obfuscation : les fonctions et les passes. Notons que nous utilisons le terme « fonction » pour désigner les fonctions du programme à obfusquer, tandis que les λ-abstractions seront désigné par le terme « abstraction ». Nous verrons enfin le système

de types utilisant le typage graduel du langage SCOL sans ses extensions (elles seront présentées en 7.3).

7.2.2 Type des fonctions

Propriétés

Chacune des fonctions du programme dans le langage possède un ensemble de propriétés d'obfuscation. Ces propriétés sont des informations qui permettent d'évaluer la protection du programme sur des points précis ou contre des attaques particulières. Les propriétés sont booléennes (par exemple : « la constante X n'est pas présente » ou « la structure de l'algorithme est protégée à un niveau élevé »). Ces propriétés sont contenues dans le type des éléments de programme et les passes changent les valeurs de certaines propriétés quand elles sont appliquées.

Ces propriétés peuvent être définies par le développeur du script de compilation simplement. Il s'agit en effet de valeurs désignées uniquement par leurs noms, il n'y a pas de construction spécifique dans le langage pour créer ou déclarer des propriétés.

Rangées

Pour gérer cet ensemble de propriétés dans le script SCOL, le type des fonctions du programme à protéger sera sous la forme de rangées contenant des propriétés.

Le type d'une fonction du programme cible dans un script SCOL aura donc la forme :

$$\langle p_1 : \pi_1; ...; p_n : \pi_n; \partial\pi \rangle$$

où les p_i sont des propriétés de la fonction et les π_i sont les présences (Pre, Abs, $\top$) et $\top$ est la présence par défaut pour les propriétés non précisées explicitement.

Contrairement aux rangées que nous avons définies en 4.3.4, nous n'associons pas de type à la présence Pre. En effet, les propriétés ne sont pas associées à un type, on s'intéresse seulement à la présence ou l'absence de la propriété.

La présence par défaut d'une propriété d'une fonction dans un script SCOL sera généralement $\top$ (ou parfois Abs). En effet, les propriétés sont des informations ajoutées sur la fonction, et si on ne les précise pas explicitement, c'est généralement qu'elles ne sont pas connues ou qu'elles sont absentes. La possibilité d'utiliser Pre comme présence par défaut

est tout de même laissée ouverte aux développeurs de scripts SCOL.

Les développeurs de scripts SCOL définissent quelles sont les propriétés présentes dans un script, donc la taille de l'ensemble des propriétés (et donc celle des rangées) est non bornée, mais l'ensemble est **fini**. Cela implique que la notation avec la présence par défaut ne serait pas obligatoire dans le langage, mais une simple facilité d'écriture, car on pourrait définir explicitement la présence de chaque propriété. Cela nous permettra également de travailler, dans la suite de ce chapitre, sur une rangée où les présences de toutes les propriétés sont définies explicitement plutôt que sur la forme normale présentée en 4.3.4.

Pour calculer une rangée après avoir ajouté ou retiré une propriété, nous définissons les opérateurs $\oplus$ et $\ominus$ [1] (les modifications sur la rangée sont notées en bleu) :

$$\underset{i\in I}{;}\ a_i :\ \pi_i \,;\partial\pi \oplus a \stackrel{\text{def}}{=} \begin{cases} \underset{i\in I}{;}\ a_i :\ \pi_i \,;\partial\pi & \text{si } a \notin \{a_i\}_{i\in I} \wedge \pi = \mathsf{Pre} \\ \underset{i\in I\setminus\{j\}}{;}\ a_i :\ \pi_i \,;\partial\pi & \text{si } a_j = a \wedge \pi = \mathsf{Pre} \\ \underset{i\in I}{;}\ a_i :\ \pi_i \,;a :\ \mathsf{Pre} \,;\partial\pi & \text{si } a \notin \{a_i\}_{i\in I} \wedge \pi \neq \mathsf{Pre} \\ \underset{i\in I\setminus\{j\}}{;}\ a_i :\ \pi_i \,;a :\ \mathsf{Pre} \,;\partial\pi & \text{si } a_j = a \wedge \pi \neq \mathsf{Pre} \end{cases}$$

$$\underset{i\in I}{;}\ a_i :\ \pi_i \,;\partial\pi \ominus a \stackrel{\text{def}}{=} \begin{cases} \underset{i\in I}{;}\ a_i :\ \pi_i \,;\partial\pi & \text{si } a \notin \{a_i\}_{i\in I} \wedge \pi = \mathsf{Abs} \\ \underset{i\in I\setminus\{j\}}{;}\ a_i :\ \pi_i \,;\partial\pi & \text{si } a_j = a \wedge \pi = \mathsf{Abs} \\ \underset{i\in I}{;}\ a_i :\ \pi_i \,;a :\ \mathsf{Abs} \,;\partial\pi & \text{si } a \notin \{a_i\}_{i\in I} \wedge \pi \neq \mathsf{Abs} \\ \underset{i\in I\setminus\{j\}}{;}\ a_i :\ \pi_i \,;a :\ \mathsf{Abs} \,;\partial\pi & \text{si } a_j = a \wedge \pi \neq \mathsf{Abs} \end{cases}$$

Informellement, l'opérateur $\oplus$ signifie que l'on indique que la propriété prend la présence Pre dans la rangée, quelle que soit sa présence précédemment. À l'inverse, l'opérateur $\ominus$ indique que la propriété prend la présence Abs dans la rangée, quelque soit sa présence précédemment. Ces opérateurs permettent de décrire facilement et intuitivement la modification de quelques propriétés dans une rangée.

Cohérence de types rangée

Dans la suite de ce chapitre, nous allons avoir besoin de déterminer si deux fonctions ont des types cohérents, c'est-à-dire qu'il n'y a pas de propriétés qui ont des présences contradictoires. On notera $\tau_1 \sim \tau_2$. Deux fonctions ont des types cohérents, si et seulement

1. Ces opérateurs et leurs définitions ne sont pas standard. Il s'agit de sucre syntaxique. Nous les décrivons ici afin de faciliter l'écriture de certaines règles par la suite.

si pour chaque propriété (qu'elle soit définie explicitement ou non) les présences dans les deux types sont, soit identique, soit au moins l'une d'entre elles est $\top$.

$$\underset{i\in I}{;}\ a_i :\ \pi_i\,;\partial\pi \sim \underset{j\in J}{;}\ a_j :\ \pi'_j\,;\partial\pi'$$

$$\stackrel{\text{def}}{=} \forall x \in \mathbb{A} \begin{cases} \pi_x = \pi'_x \ \vee\ \pi_x = \top \ \vee \pi'_x = \top & \text{si } x \in I \cap J \\ \pi_x = \pi' \ \vee\ \pi_x = \top \ \vee \pi' = \top & \text{si } x \in I \setminus J \\ \pi = \pi'_x \ \vee\ \pi = \top \ \vee \pi'_x = \top & \text{si } x \in J \setminus I \\ \pi = \pi' \ \vee\ \pi = \top \ \vee \pi' = \top & \text{si } x \notin I \cup J \end{cases}$$

Intuitivement, le fait que deux types de fonctions soient cohérents signifie qu'ils sont identiques du point de vue des propriétés pour lesquelles on a des informations.

Sous-typage des rangées

Nous définissons également le sous-typage de rangée, qui permettra de déterminer si une rangée est un sous-type d'une autre rangée. Intuitivement, une rangée sera sous type d'une autre rangée si la présence de chaque propriété de la première est sous type de la présence de la propriété dans la seconde. Les règles de sous typage pour les présences sont les suivantes :

$$\mathsf{Abs} <: \mathsf{Abs} \qquad \mathsf{Pre} <: \mathsf{Pre} \qquad \top <: \top \qquad \mathsf{Abs} <: \top \qquad \mathsf{Pre} <: \top$$

Et la définition mathématique du sous-typage de rangée est donc la suivante :

$$\underset{i\in I}{;}\ a_i :\ \pi_i\,;\partial\pi <: \underset{j\in J}{;}\ a_j :\ \pi'_j\,;\partial\pi'$$

$$\stackrel{\text{def}}{=} \forall x \in \mathbb{A} \begin{cases} \pi_x <: \pi'_x & \text{si } x \in I \cap J \\ \pi_x <: \pi' & \text{si } x \in I \setminus J \\ \pi <: \pi'_x & \text{si } x \in J \setminus I \\ \pi <: \pi' & \text{si } x \notin I \cup J \end{cases}$$

Nous allons voir dans la suite de ce chapitre comment les règles de sous typage nous permettent d'écrire des scripts et de définir des passes génériques.

7.2.3 Type des passes

Forme du type

Dans SCOL, une passe est sémantiquement similaire à une abstraction : elle prend une expression en entrée et donne une autre expression en sortie. La différence avec une abstraction est que le calcul de la valeur de sortie est implémenté en dehors du langage. Une passe aura donc un type similaire aux abstractions, la forme du type des passes sera donc :

$$e ::= ... \mid \texttt{let\_pass}\, x : \tau\, \texttt{from}\, \text{PASS}\, \texttt{in}\, e \qquad\qquad \tau_r ::= ... \mid \tau_r \to \tau_r$$

Le type d'une passe aura donc la même forme que celui des abstractions. Le type du paramètre correspondant à l'entrée sera en général un type correspondant à une fonction telle que présentée dans la section précédente. Une partie des passes aura pour type d'entrée un type tuple contenant des types de fonctions et d'autres types pour les différents arguments d'une fonction (par exemple : $\langle p_1 : \pi_1; ...; \partial\pi \rangle * \texttt{bool} * \texttt{int}$). De même, pour le type de sortie d'une passe, il s'agira en général de type de fonction ou de type tuple contenant des fonctions et autres paramètres. Ce type correspond à la forme des entrées et sorties des passes que nous avons définie en 5.2.3.

Les passes d'analyses peuvent être représentées par des types de cette forme, avec un type de sortie qui sera un booléen, un entier ou une autre valeur représentant le résultat de l'analyse.

La forme du type des passes étant la même que celle des abstractions, la règle de type concernant l'application des passes est également la même que celle de l'application d'abstraction. On rappelle qu'il s'agit de :

$$\textsc{TApp}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 @ e_2 : \tau_2}$$

Cette forme de type ($\tau_1 \to \tau_2$) permet également de décrire les changements de propriétés d'une fonction lorsqu'on exécute une passe. Le type des passes contient le type des fonctions qu'elles prennent en entrée et les types des fonctions en sortie (c'est-à-dire une fois transformée par la passe). Ces types de fonctions contiennent les informations de propriétés, donc le type des passes indique les transformations sur les propriétés des

fonctions.

Par exemple, on peut décrire la passe de *bogus control flow* (cf. 2.5.1) avec le type suivant :

$$\langle \text{CFProt} : \mathsf{Abs}; \partial\top \rangle \to \langle \text{CFProt} : \mathsf{Pre}; \partial\top \rangle$$

Ce type signifie que la passe de *bogus control flow* prend en paramètre une fonction dont le flot de contrôle n'est pas protégé (la propriété CFProt a la présence Abs) et transforme la fonction pour protéger son flot de contrôle (le type de la sortie est une fonction dans laquelle la propriété CFProt a la présence Pre).

Prenons une passe d'analyse comme un second exemple. Cette passe va déterminer si une fonction contient des constantes (car si c'est le cas on pourra vouloir les protéger avec différentes techniques d'obfuscation). Son type pourra être écrit :

$$\langle \partial\top \rangle \to \texttt{bool}$$

Il s'agit d'une passe prenant en entrée une fonction quelconque et donnant en sortie un booléen contenant l'information sur la présence de constante dans la fonction.

Sous typage pour l'application de passes

Avec la règle de typage TApp et le type des passes tels que nous venons de les décrire, la fonction d'entrée doit avoir exactement le type de l'entrée de la passe : la présence de chaque propriété doit être exactement la même, y compris pour la valeur $\top$.

Par exemple, la passe de type $\langle \text{CFProt} : \mathsf{Abs}; \partial\top \rangle \to \langle \text{CFProt} : \mathsf{Pre}; \partial\top \rangle$ ne pourra pas être appliquée sur une fonction de type $\langle \text{CFProt} : \mathsf{Abs}; \text{AProp} : \mathsf{Pre}; \partial\top \rangle$.

Or, l'intérêt principal du choix de l'utilisation des rangées et de la possibilité d'utiliser la présence $\top$ est de permettre de décrire des types de fonctions avec des informations (propriétés) qui sont inconnues sans que cela affecte le script d'obfuscation lorsque cette information n'est pas nécessaire.

Pour pallier cette limitation, nous modifions la règle TApp de la façon suivante :

$$\textsc{TAppII}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_3 \qquad \Gamma \vdash \tau_3 <: \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Avec cette règle, une passe dont le type d'entrée fixe des propriétés à $\top$ (c'est-à-dire que la connaissance de cette propriété n'est pas nécessaire à l'exécution de la passe) peut quand même être appliquée sur des fonctions dont ces propriétés sont à Pre ou Abs.

Utilisation de contraintes

Avec les types et les règles de typage que nous avons décrits jusqu'à présent il peut y avoir une perte d'information lors de l'application de passe. En effet, reprenons la passe de *bogus control flow* de type $\langle \text{CFProt} : \mathsf{Abs}; \partial\top \rangle \to \langle \text{CFProt} : \mathsf{Pre}; \partial\top \rangle$ que nous avons utilisée précédemment. On peut supposer que cette passe ne modifie pas les autres propriétés de la fonction sur laquelle elle est appliquée (autre que CFProt). Or, d'après la règle de typage, si on applique cette passe à une fonction ayant pour type $\langle \text{CFProt} : \mathsf{Abs}; \text{AProp} : \mathsf{Pre}; \partial\top \rangle$, le type de la fonction résultante de l'application sera : $\langle \text{CFProt} : \mathsf{Pre}; \partial\top \rangle$. Il y a donc une perte d'information, on a perdu l'information sur la présence de la propriété AProp.

Nous souhaitons permettre de décrire une passe dont le type du résultat sera dépendant du type d'entrée. Cela permettra en particulier d'écrire que le type de la fonction résultante de l'application est celui de la fonction avec des ajouts ou retraits de propriétés, et donc d'éviter la perte d'information lors de l'application de passe. Pour cela nous allons utiliser des types contraints [168][149][3]. Le type d'une passe pourra être une contrainte qui lie le type de la sortie de la passe au type de l'entrée.

La forme des types contraints sera la suivante :

$$\tau ::= \tau_r \mid \tau_c \,\texttt{with}\, C_{main} \qquad \tau_c ::= \tau_r \mid \alpha \qquad C_{main} ::= C_{main} \vee C_{main} \mid C$$

$$C ::= C \wedge C \mid \tau <: \tau \mid \tau = \tau \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\, C \mid (C)$$

α est une variable de type, elle sera utilisée pour lier les entrées et les sorties dans les types contraints. Le type $\tau \,\texttt{with}\, C_{main}$ représente les types contraints. Une contrainte est une disjonction de conjonctions de proposition logique sur les types. Ces propositions peuvent être une relation de sous typage, une égalité de type ou une négation.

Grâce aux contraintes on peut maintenant écrire le type de la passe de *bogus control flow* présentée précédemment sous la forme :

$$\alpha \to \beta \,\texttt{with}\, (\alpha <: \langle \text{CFPROT} : \mathsf{Abs}; \partial\top\rangle) \wedge \beta = \alpha \oplus \text{CFPROT}$$

Ce type signifie que la passe doit être appliquée à une expression de type α qui doit respecter la contrainte $\alpha <: \langle \text{CFPROT} : \mathsf{Abs}; \partial\top\rangle$. Le résultat de l'application de cette passe sera de type β qui respecte la contrainte $\beta = \alpha \oplus \text{CFPROT}$, c'est-à-dire, qu'il s'agit du type de la fonction d'entrée avec la présence de la propriété CFPROT fixée à Pre. Donc dans le cas où l'entrée est la fonction que nous avons présentée précédemment de type $\langle \text{CFPROT} : \mathsf{Abs}; \text{APROP} : \mathsf{Pre}; \partial\top\rangle$, la sortie sera de type $\langle \text{CFPROT} : \mathsf{Pre}; \text{APROP} : \mathsf{Pre}; \partial\top\rangle$. On ne perd plus l'information de type sur la propriété APROP .

Prenons un autre exemple, une passe qui protège le programme si celui-ci possède la propriété TOPROT et ne fait rien sinon (dans le cas où TOPROT est à Pre elle agit, s'il est à Abs ou $\top$ elle ne fait rien) aura le type suivant :

$$\alpha \to \beta \,\texttt{with}\, (\alpha <: \langle \text{TOPROT} : \mathsf{Pre}; \partial\top\rangle) \wedge \beta = \alpha \ominus \text{TOPROT} \bigvee$$

$$(\alpha <: \langle \text{TOPROT} : \mathsf{Abs}; \partial\top\rangle) \wedge \beta = \alpha \bigvee$$

$$(\alpha <: \langle \text{TOPROT} : \top; \partial\top\rangle) \wedge \beta = \alpha$$

Cet exemple montre une utilisation possible de la disjonction de type dans les contraintes.

Solutions des contraintes

Une contrainte C contenant les variables de types $\{\alpha_i\}_{[1,n]}$ est bien formée ($wf(C)$), si et seulement si, $\forall i \in [1,n], \exists < row_i >, wf([\alpha_i \mapsto < row_i >]C)$. Si une contrainte C ne contient pas de variable de type, elle est bien formée si et seulement si, C est une formule valide. Autrement dit, une contrainte C est bien formée, si et seulement si, elle est satisfiable. Informellement, cela signifie qu'une contrainte est bien formée s' il existe un ensemble de type fonction qui résout la contrainte.

Un type τ est une solution de la contrainte C pour la variable α notée $\tau \in S[C](\alpha)$ si et seulement si, en substituant α par τ dans l'équation logique formant la contrainte, la contrainte est bien formée : $wf([\alpha \mapsto \tau]C)$.

L'ensemble des solutions à une contrainte C pour la variable α est noté $S[C](\alpha)$. Cet ensemble est non-vide pour une contrainte bien formée, car une contrainte bien formée possède au moins une solution par définition. Cet ensemble est fini. En effet, les solutions aux contraintes sont des types de fonctions, or l'ensemble des types de fonctions est fini, car l'ensemble des propriétés est fini (donc l'ensemble des rangées distinctes formées à partir de ces propriétés est fini). Soit N le nombre de propriétés, on peut former 3^N rangées distinctes à partir de ces propriétés (pour chaque propriété, on choisit entre les présences Pre, Abs et $\top$). Donc la taille de l'ensemble des solutions $S[C](\alpha)$ est minorée par 1 et majoré par 3^N.

Cohérence de contraintes

Comme notre langage utilise le typage graduel, nous avons également besoin de définir la cohérence de type pour les types définis avec des contraintes. On distingue deux cas : soit un seul des deux types est un type contraint, soit les deux types sont contraints.

Si un seul des deux types est contraint : soit τ le type contraint et τ' le type non contraint. Soit S l'ensemble des solutions aux contraintes sur τ.

$$\tau \sim \tau' \Leftrightarrow \exists \tau_s \in S, \tau_s \sim \tau'$$

Si les deux types sont contraints : soit S l'ensemble des solutions aux contraintes sur τ et S' l'ensemble des solutions aux contraintes sur τ'.

$$\tau \sim \tau' \Leftrightarrow S \cap S' \neq \varnothing$$

7.2.4 SCOL typé graduellement

Maintenant que nous avons défini les types des programmes et des passes, nous pouvons définir le système de types de SCOL.

Syntaxe de SCOL et SCOL$_{cast}$

La syntaxe de SCOL typé est très similaire à celle de SCOL non typé (SCOL$_{sta}$), à la différence que l'on ajoute l'information de type dans l'expression `let_pass` et les types que nous avons présentés précédemment. La syntaxe de SCOL est donc :

$$
\begin{array}{lll}
main & ::= & \texttt{script}(x : \tau) = e \\
e & ::= & x \mid \lambda x : \tau_r.e \mid e \mathbin{@} e \mid \texttt{let\_pass}\, x : \tau_r\, \texttt{from}\, \textsc{pass}\, \texttt{in}\, e \\
\tau & ::= & \tau_r \mid ? \\
\tau_r & ::= & \tau_r \to \tau_r \mid \alpha \mid \tau_r\, \texttt{with}\, C_{main} \mid < row > \\
row & ::= & \textsc{Prop} : pres; row \mid \partial pres \\
pres & ::= & \mathsf{Pre} \mid \mathsf{Abs} \mid \top \\
C_{main} & ::= & C_{main} \bigvee C_{main} \mid C \\
C & ::= & C \wedge C \mid \tau_r <: \tau_r \mid \tau_r = \tau_r \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\, C \mid (C)
\end{array}
$$

SCOL est un langage typé graduellement, lors de la compilation de script SCOL il y aura donc des insertions de *cast*. Cependant, nous avons limité le type des abstractions et des passes à des types non graduels (τ_r). Dans le cadre des scripts d'obfuscation, permettre l'utilisation d'abstraction et de passe typée dynamiquement ajoute beaucoup de complexité au langage au regard de ce que cela apporte. En effet, la description des passes, décrites en 5.2, demande la connaissance des entrées/sorties des passes, et donc de leur type. De plus, dans notre processus d'obfuscation, les abstractions ne peuvent être définies que statiquement dans SCOL. Vues autrement, les abstractions définies dynamiquement sont les passes. L'intérêt d'autoriser le typage dynamique pour les abstractions et les passes est donc faible par rapport à la complexité ajoutée.

Nous définissons la syntaxe de SCOL$_{cast}$, le langage vers lequel SCOL est compilé et qui contient les insertions de *cast*. La syntaxe de SCOL$_{cast}$ est basée sur celle de SCOL$_{sta}$ (cf.6.2.1) à laquelle on ajoute les *cast* :

$$
\begin{array}{lll}
main_{cast} & ::= & \texttt{script}(x) = e_{cast} \\
e_{cast} & ::= & x \mid \lambda x.e_{cast} \mid e_{cast} \mathbin{@} e_{cast} \mid \texttt{let\_pass}\, x\, \texttt{from}\, \textsc{pass}\, \texttt{in}\, e_{cast} \mid (< \tau >\ e_{cast})
\end{array}
$$

On note que l'information de type n'est plus présente dans l'expression `let_pass`. Cette information n'est pas utile, car les calculs de types ont déjà été effectués. Notons également que nous avons omis certains indices qui étaient présents dans SCOL$_{sta}$ pour alléger l'écriture des termes, car il n'y a pas de confusion possible ici.

Enfin la sémantique de SCOL$_{cast}$ est identique à celle de SCOL$_{sta}$ vu dans en 6.2.3, à laquelle on ajoute la sémantique du *cast* décrite en page 128. La sémantique complète est disponible en B.

Règle de compilation vers SCOL$_{cast}$

Nous allons maintenant décrire les règles de compilation de SCOL vers SCOL$_{cast}$, il s'agira en même temps des règles de typage de l'expression.

Cette compilation sera notée $e \Rightarrow e_{cast} : \tau$ où e et e_{cast} sont les expressions dans SCOL et SCOL$_{cast}$ respectivement, et τ le type de e et e_{cast}.

Commençons par les règles de compilation du λ-calcul de SCOL :

$$\textsc{ScriptObf}\ \frac{x : \tau \vdash e \Rightarrow e_{cast} : \tau'}{\texttt{script}(x : \tau) = e \Rightarrow \texttt{script}(x) = e_{cast} : \tau'}$$

$$\textsc{App1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'' \qquad \tau'' <: \tau}{\Gamma \vdash e_1 \,@\, e_2 \Rightarrow {e_{cast}}^1 \,@\, {e_{cast}}^2 : \tau'}$$

$$\textsc{App2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'' \qquad \tau'' \sim \tau \qquad \neg(\tau'' <: \tau)}{\Gamma \vdash e_1 \,@\, e_2 \Rightarrow {e_{cast}}^1 \,@\, ((< \tau >\ {e_{cast}}^2)) : \tau'}$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow x : \tau} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow e_{cast} : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \lambda x . e_{cast} : \tau_1 \to \tau_2}$$

La règle ScriptObf est très similaire aux règles d'abstractions, car cette construction possède une sémantique similaire à un `let` usuel. Les règles App1, App2, Var et Lam sont les règles usuelles du λ-calcul typé graduellement avec sous-typage.

On ajoute à ces règles la règle de compilation de la construction `let_pass`, qui est similaire à une règle de typage de `let` usuelle.

$$\textsc{LetPass}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow e_{cast} : \tau_2}{\Gamma \vdash \texttt{let\_pass}\, x : \tau_1 \,\texttt{from}\, \text{PASS} \,\texttt{in}\, e \Rightarrow \texttt{let\_pass}\, x \,\texttt{from}\, \text{PASS} \,\texttt{in}\, e_{cast} : \tau_2}$$

Enfin, il y a deux règles supplémentaires pour l'application, lorsque le type de l'abstraction (ou du `let_pass`) est un type contraint. On note $S[C](\alpha)$ l'ensemble des solutions de la contrainte C pour le type α.

$$\textsc{App3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau_1 \to \tau_2\ \texttt{with}\ C \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau \qquad \tau <: \tau' \qquad \tau' \in S[C](\tau_1)}{\Gamma \vdash e_1 \mathbin{@} e_2 \Rightarrow {e_{cast}}^1 \mathbin{@} {e_{cast}}^2 : \tau_2\ \texttt{with}\ [\tau_1 \mapsto \tau]C}$$

$$\textsc{App4}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau_1 \to \tau_2\ \texttt{with}\ C \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau \qquad \nexists \tau_3, \tau <: \tau_3, \tau_3 \in S[C](\tau_1) \qquad \exists\, \tau' \in S[C](\tau_1), \tau \sim \tau'}{\Gamma \vdash e_1 \mathbin{@} e_2 \Rightarrow {e_{cast}}^1 \mathbin{@} (< \tau_1\ \texttt{with}\ C >\ {e_{cast}}^2) : \tau_2\ \texttt{with}\ [\tau_1 \mapsto \tau]C}$$

La règle App3 fixe dans la contrainte la variable de type de l'entrée de l'abstraction, par le type de l'expression sur laquelle elle est appliquée. Pour que ces expressions soient bien typées, il est nécessaire que le type contraint possède une solution et que le type de l'entrée soit le sous-type de l'une de ces solutions.

Si ce n'est pas le cas, alors la règle App4 indique que s'il existe une solution à la contrainte qui est cohérente avec le paramètre d'entrée, on insère un *cast* dans l'application.

Exemples

Prenons l'exemple de script suivant :

$\texttt{script}(x : \partial\top) = ($
$\quad \texttt{let\_pass}\ obf : \alpha \to \beta\ \texttt{with}\ (\alpha <: \textsc{Pobf}\ :\ \mathsf{Abs}; \partial\top) \wedge (\beta = \alpha \oplus \textsc{Pobf})\ \texttt{from}\ \textsc{obf}\ \texttt{in}$
$\quad obf\ @_{sta}\ x)$

Ce script prend un programme dynamique (ces propriétés sont toutes à $\top$) en argument. Ensuite il applique la passe *obf* sur ce programme. Cette passe s'applique sur un programme dont la propriété Pobf est absente, et le résultat aura cette propriété présente $\beta = \alpha \oplus \textsc{Pobf}$.

Comme l'entrée du script n'est pas une solution des contraintes de la passe, un cast sera inséré (règle App4). La traduction de ce script sera donc :

$$\begin{array}{l}\texttt{script}(x) = (\\ \quad \texttt{let\_pass}\ \mathit{obf}\ \texttt{from}\ \ \textsc{obf}\ \ \texttt{in}\\ \quad \mathit{obf}\ @_{sta}\ (< \alpha\ \texttt{with}\ (\alpha <: \textsc{Pobf}\ :\ \mathsf{Abs}; \partial\top) \wedge (\beta = \alpha \oplus \textsc{Pobf}) >\ x))\end{array}$$

Si on modifie légèrement cet exemple en modifiant le type d'entrée du script de façon à ce qu'il prenne maintenant un programme dans lequel toutes les propriétés ont la présence Pre. Le script d'obfuscation devient le suivant :

$$\begin{array}{l}\texttt{script}(x : \partial\,\mathsf{Pre}) = (\\ \quad \texttt{let\_pass}\ \mathit{obf} : \alpha \to \beta\ \texttt{with}\ (\alpha <: \textsc{Pobf}\ :\ \mathsf{Abs}; \partial\top) \wedge (\beta = \alpha \oplus \textsc{Pobf})\ \texttt{from}\ \ \textsc{obf}\ \ \texttt{in}\\ \quad \mathit{obf}\ @_{sta}\ x)\end{array}$$

Dans ce script, l'application de *obf* sur x n'est pas bien typée, donc le script n'est pas bien typé. En effet, le type de x n'est pas un sous-type d'une solution à la contrainte de *obf* (comme exigé par la règle App3), car la propriété Pobf est à Pre dans x. Notre système de types détecte statiquement que ce script d'obfuscation ne pourra jamais être appliqué.

Sémantique d'évaluation et de traduction du *cast*

Pour utiliser le langage graduel, nous avons défini une nouvelle construction dans le langage SCOL_{cast} qui n'était pas définie dans SCOL_{sta} : le *cast*. Nous devons donc définir également les sémantiques de cette construction.

Sémantique d'évaluation : il y a deux cas pour l'évaluation du *cast*. Si l'expression a une relation de sous-typage avec le type du *cast*, alors le cast s'évalue comme l'expression castée (règle Cast). Si, par contre, le type n'est pas un sous-type du type du *cast*, alors le résultat de l'évaluation sera une erreur (règle CastError).

$$\textsc{Cast}\ \frac{e_w \Rightarrow_w v \qquad \texttt{typeof}(e_w) = \tau \qquad \tau <: \tau'}{(< \tau' >\ e_w) \Rightarrow_w v}$$

$$\textsc{CastError}\ \frac{\texttt{typeof}(e_w) = \tau \qquad \neg(\tau <: \tau')}{(< \tau' >\ e_w) \Rightarrow_w \textsc{Error}}$$

Pour définir ces règles, nous avons également défini la fonction `typeof`(). Cette fonction renvoie le type réel d'une expression, c'est-à-dire le type de l'expression à l'exécution du script. Cela rend la sémantique d'évaluation dépendante des types. Donc, à l'exécution du programme, nous devrons conserver les types réels des termes pour réaliser ces *casts*

Sémantique de traduction : la construction *cast* est ajoutée dans les constructions utilisant les règles de contextes. La syntaxe de SCOL est la suivante :

$$S_w[\_] ::= ... \mid (< \tau > \ e_w)$$

La traduction du *cast* se fait donc exclusivement avec la règle de contexte Context. Le *cast* étant, par définition, une opération purement dynamique, il n'y pas de règles d'évaluation supplémentaires pour l'évaluation statique.

7.2.5 Propriétés du système de types

Notre système de types a été conçu avec comme objectif d'assurer certaines propriétés de sûreté du langage. En particulier, nous souhaitons que le système de types assure les propriétés de progression et de préservation. C'est-à-dire qu'un terme bien typé peut être évalué (progression) et qu'un terme bien typé est évalué en un terme bien typé (préservation). De plus, le système de types doit assurer que les différentes contraintes d'application des passes sont respectées.

Informellement, avec le typage graduel, nous souhaitons également assurer que toutes les erreurs de types détectables statiquement sont bien détectées statiquement et que les autres sont détectées dynamiquement. Nous souhaitons également que tous les scripts valides soient bien typés et ne soient pas rejetés par le système de types (statique ou dynamique).

Nous n'avons pas prouvé que notre système de types assure ces propriétés. Dans le futur, il sera important de vérifier que ces propriétés sont respectées, car une grande partie de l'intérêt de SCOL repose sur les propriétés de ce système de types.

7.3 Typage des extensions à SCOL

Maintenant que nous avons défini le système de types de langage SCOL de base, nous allons définir les systèmes de types pour les extensions au langage que nous avons

présenté en 6.3. Nous ne présenterons pas ici la syntaxe et la sémantique dans SCOL_{cast} de ces extensions, car elles sont identiques à celle de SCOL que nous avons présenté précédemment. La syntaxe et la sémantique complète sont disponibles en annexes A et B.

7.3.1 Booléens et structures conditionnelles

Pour les constructions booléennes nous avons d'abord besoin d'ajouter un type `bool` aux types du langage : $\tau ::= ... \mid \texttt{bool}$.

Nous ajoutons également des règles de compilation et de typage pour les nouvelles constructions. Nous avons d'abord les règles triviales pour les constantes et les opérations logiques :

$$\textsc{True}\ \Gamma \vdash \texttt{true} \Rightarrow \texttt{true} : \texttt{bool} \qquad \textsc{False}\ \Gamma \vdash \texttt{false} \Rightarrow \texttt{false} : \texttt{bool}$$

$$\textsc{Not1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e \Rightarrow \texttt{not}\ e_{cast} : \texttt{bool}}$$

$$\textsc{Not2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{not}\ e \Rightarrow \texttt{not}(<\texttt{bool}>\ e_{cast}) : \texttt{bool}}$$

$$\textsc{Binop1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{bool}}{\Gamma \vdash e_1\, binop\, e_2 \Rightarrow {e_{cast}}^1\, binop\, {e_{cast}}^2 : \texttt{bool}}$$

$$\textsc{Binop2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{bool}}{\Gamma \vdash e_1\, binop\, e_2 \Rightarrow (<\texttt{bool}>\ {e_{cast}}^1)\, binop\, {e_{cast}}^2 : \texttt{bool}}$$

Les règles True et False indiquent le type des constantes booléennes. Les opérandes des opérateurs logiques doivent être des booléens, si cela est le cas statiquement (Not1 et Binop1) l'opération est bien typée, s'ils sont typés dynamiquement (Not2 et Binop2), un *cast* est nécessaire. Il existe bien sûr les règles symétriques pour le typage graduel des opérateurs binaires qui sont décrites en annexe D.

Les règles de compilation et de typage de la structure conditionnelle sont les suivantes :

$$\textsc{If}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \quad \Gamma \vdash e_3 \Rightarrow {e_{cast}}^3 : \tau_2 \quad \exists \tau, \tau_1 <: \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \Rightarrow \texttt{if}\, {e_{cast}}^1\, \texttt{then}\, {e_{cast}}^2\, \texttt{else}\, {e_{cast}}^3 : \tau}$$

$$\textsc{If1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \quad \Gamma \vdash e_3 \Rightarrow {e_{cast}}^3 : \tau_2 \quad \exists \tau, \tau_1 <: \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \Rightarrow \texttt{if}\, (< \texttt{bool} >\ {e_{cast}}^1)\, \texttt{then}\, {e_{cast}}^2\, \texttt{else}\, {e_{cast}}^3 : \tau}$$

$$\textsc{If2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \quad \Gamma \vdash e_3 \Rightarrow {e_{cast}}^3 : \tau_2 \quad \exists \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \Rightarrow \texttt{if}\, {e_{cast}}^1\, \texttt{then}\, (< \tau >\ {e_{cast}}^2)\, \texttt{else}\, {e_{cast}}^3 : \tau}$$

$$\textsc{If3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \quad \Gamma \vdash e_3 \Rightarrow {e_{cast}}^3 : ? \quad \exists \tau, \tau_1 <: \tau}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \Rightarrow \texttt{if}\, {e_{cast}}^1\, \texttt{then}\, {e_{cast}}^2\, \texttt{else}\, (< \tau >\ {e_{cast}}^3) : \tau}$$

$$\textsc{If23}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \quad \Gamma \vdash e_3 \Rightarrow {e_{cast}}^3 : ?}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \Rightarrow \texttt{if}\, {e_{cast}}^1\, \texttt{then}\, {e_{cast}}^2\, \texttt{else}\, {e_{cast}}^3 : ?}$$

La règle If décrit la règle de typage pour les cas où le type de toutes les expressions n'est pas graduel, le résultat de la structure conditionnel est un sur type des expressions la composant. Dans la règle If1, l'expression de la condition est typée dynamiquement, elle sera donc *cast* en `bool` dans SCOL$_{cast}$. Les autres règles (If2, If3 et If23) concernent les cas où l'une ou les deux expressions de sortie de la structure conditionnelle sont typé graduellement et l'insertion de *cast* est nécessaire.

Enfin, à la fois la condition et les résultats peuvent être typés graduellement, nous n'avons pas écrit les règles ici, car il s'agit simplement de la fusion des autres règles [2].

7.3.2 Nombres entiers

Pour les nombres entiers, les règles de typages sont assez simples. D'abord le type `int` est ajouté : $\tau ::= ... \mid \texttt{int}$. Ensuite, on distingue les opérateurs sur les entiers en deux catégories : les opérateurs de comparaisons $cop ::= < \mid <= \mid > \mid = \mid \leq \mid \geq \mid \neq$, et les opérateurs arithmétiques $arithOp ::= + \mid - \mid / \mid * \mid \texttt{mod}$. Nous séparons ces opérateurs pour le

2. l'ensemble des règles est décrit en annexe D.

système de types (contrairement à la sémantique), car les règles de types sont différentes (cf. règles ci-dessous). Les règles de compilation et de typage sont les suivantes :

$$\textsc{Int}\ \Gamma \vdash int \Rightarrow int : \texttt{int} \qquad \textsc{Cop1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, cop\, e_2 \Rightarrow {e_{cast}}^1\, cop\, {e_{cast}}^1 : \texttt{bool}}$$

$$\textsc{Cop2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, cop\, e_2 \Rightarrow (<\texttt{int}>\ {e_{cast}}^1)\, cop\, {e_{cast}}^1 : \texttt{bool}}$$

$$\textsc{ArithOp1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, arithOp\, e_2 \Rightarrow {e_{cast}}^1\, arithOp\, {e_{cast}}^1 : \texttt{int}}$$

$$\textsc{ArithOp2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, arithOp\, e_2 \Rightarrow (<\texttt{int}>\ {e_{cast}}^1)\, arithOp\, {e_{cast}}^1 : \texttt{int}}$$

De façon similaire aux règles concernant les booléens, la règle Int indique le type des constantes entières. Les opérandes des opérateurs doivent être des entiers, il y a aura donc des *cast* si certaines expressions sont typées dynamiquement (Cop2 et ArithOp2) et elles seront directement bien typé si les types statiques sont `int` (Cop1 et ArithOp1). Il existe des symétries pour certaines règles présentées ici, l'ensemble des règles est disponible en annexe D.

7.3.3 Listes

Types des listes

Les listes contiennent une séquence d'éléments qui sont de même type. Le type d'une liste contiendra donc l'information sur le fait qu'il s'agisse d'une liste et sur le type contenue dans la liste. Nous le notons : $\tau ::= ... \mid \tau\,\texttt{list}$

Nous définissons également les règles de sous typage et de cohérence sur les types de listes :

$$\tau\texttt{list} \sim \tau'\texttt{list} \Leftrightarrow \tau \sim \tau' \qquad\qquad \tau\texttt{list} <: \tau'\texttt{list} \Leftrightarrow \tau <: \tau'$$

Intuitivement, un type de liste est un sous-type (resp. cohérent avec) d'un autre si le type des éléments qui composent la première est un sous-type (resp. cohérent avec) de celui des éléments de la seconde.

Règles d'insertion de cast et de typages des listes

Nous commençons par définir les règles de typages et d'insertion de *cast* pour la construction de listes :

$$\textsc{ListEmpty}\ \Gamma \vdash [] \Rightarrow [] : ?\,\texttt{list}$$

$$\textsc{ListCons1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ?\,\texttt{list} \qquad \tau \neq ?}{\Gamma \vdash e_1 :: e_2 \Rightarrow {e_{cast}}^1 :: (< \tau\texttt{list} > \ {e_{cast}}^2) : \tau'\,\texttt{list}}$$

$$\textsc{ListCons2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'\,\texttt{list} \qquad \exists \tau'', \tau <: \tau'', \tau' <: \tau''}{\Gamma \vdash e_1 :: e_2 \Rightarrow {e_{cast}}^1 :: {e_{cast}}^2 : \tau''\,\texttt{list}}$$

$$\textsc{ListCons3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'\,\texttt{list} \qquad \tau' \neq ?}{\Gamma \vdash e_1 :: e_2 \Rightarrow (< \tau' > \ {e_{cast}}^1) :: {e_{cast}}^2 : \tau'\,\texttt{list}}$$

La liste vide [] est de type ?`list` (règle ListEmpty), c'est-à-dire qu'elle est typée dynamiquement, le type des éléments de la liste sera précisé lorsqu'on ajoute d'autres éléments à la liste (règle ListCons1), qui ajoutera un *cast* sur la liste vide. Notons qu'un *cast* d'une liste vide vers un type liste sera toujours réussi.

Sinon lorsqu'on construit la liste, son type sera un sur type commun aux éléments qui la composent et on insère des *cast* si besoin (règles ListCons2 et ListCons3).

Les règles pour les opérateurs de manipulation des listes sont les suivantes :

$$\textsc{Tl1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau\,\texttt{list}}{\Gamma \vdash \texttt{tl}\,e \Rightarrow \texttt{tl}\,e_{cast} : \tau\,\texttt{list}} \qquad \textsc{Hd1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau\,\texttt{list}}{\Gamma \vdash \texttt{hd}\,e \Rightarrow \texttt{hd}\,e_{cast} : \tau}$$

$$\textsc{Tl2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{tl}\,e \Rightarrow \texttt{tl}\,(<?\,\texttt{list} >\ e_{cast}) : ?\,\texttt{list}}$$

$$\textsc{Hd2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{hd}\,e \Rightarrow \texttt{hd}\,(<?\,\texttt{list} >\ e_{cast}) : ?}$$

Le résultat de ces opérations est une liste donc qui conserve le même type pour l'opération *tail*(Tl1), et le premier élément de la liste pour *head*, son type sera donc celui du contenu de la liste (Hd1). Ces opérations s'appliquent sur des listes, donc un *cast* vers un type générique de liste (? `list`) peut être nécessaire. Le résultat de ces opérations est une liste qui conserve le même type pour l'opération *tail*(Tl1), et le premier élément de la liste pour *head*, son type sera donc celui du contenu de la liste (Hd2 et Tl2).

7.3.4 Tuples

Type des tuples

Les tuples sont des ensembles qui contiennent des éléments de types qui peuvent être différents. Le type du tuple contiendra les informations de type de l'ensemble des éléments composant le tuple. Nous noterons le type des tuples sous la forme : $\tau ::= ... \mid \tau * ... * \tau$.

Comme pour les listes, nous définissons les relations de sous typage et de cohérence des types de tuples :

$$\tau_1 * ... * \tau_n \sim \tau'_1 * ... * \tau'_n \Leftrightarrow \forall i \in [1, n], \tau_i \sim \tau'_i$$

$$\tau_1 * ... * \tau_n <: \tau'_1 * ... * \tau'_n \Leftrightarrow \forall i \in [1, n], \tau_i <: \tau'_i$$

Intuitivement, un type de tuple est un sous-type (resp. cohérent avec) d'un autre si le type des éléments qui composent le premier est un sous-type (resp. cohérent avec) de celui des éléments du second.

Règles d'insertion de cast et de typages des listes

Nous commençons par définir les règles d'insertion de cast pour la construction de tuples :

$$\textsc{TupleCons}\ \frac{\forall i \in [1,n], \Gamma \vdash e_i \Rightarrow e_{cast}^{i} : \tau_i}{\Gamma \vdash (e_1, ..., e_n) \Rightarrow {e_{cast}}^{1}, ..., e_{cast}^{n} : \tau_1 * ... * \tau_n}$$

Le type d'un tuple est simplement la concaténation des types des éléments qui le composent (règle TupleCons).

Les règles d'insertion de cast (Proj et Proj1) pour l'opération de projection sont également très simples : le type résultant de l'opération est le type correspondant à l'élément dans le type tuple. Cependant lors de la projection d'un tuple dynamique, il est nécessaire d'ajouter une vérification dynamique sur la taille du tuple. Pour cela, nous ajoutons un type `TupleSize`(i), qui permet d'indiquer que le tuple a une taille supérieure à i. Ce type ne sera utilisé que dans les casts de la règle Proj1.

$$\textsc{Proj}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau_1 * ... * \tau_n}{\Gamma \vdash \pi_i\, e \Rightarrow \pi_i\, e_{cast} : \tau_i}$$

$$\textsc{Proj1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \pi_i\, e \Rightarrow \pi_i\, (< \texttt{TupleSize}(i) >\ e_{cast}) : ?}$$

7.3.5 Récursions

La fonction point fixe permettant de réaliser la récursion possèdera la syntaxe suivante :

$$e ::= ... \mid \mu f : \tau \rightarrow \tau . \lambda x : \tau . e$$

$$e_{cast} ::= ... \mid \mu f : \tau \rightarrow \tau . \lambda x . e_{cast}$$

La règle de typage du μ (Rec) ressemble à celle de l'abstraction, et l'application de fonctions récursives utilise les mêmes règles que les autres applications (Les règles commençant par App).

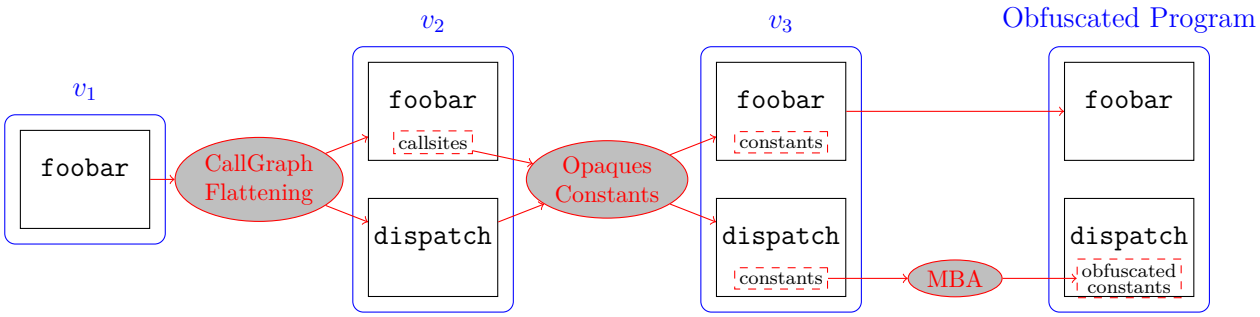

FIGURE 7.1 – Un exemple de chaîne d'obfuscation

$$\textsc{Rec}\ \frac{\Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e \Rightarrow e_{cast} : \tau_3}{\Gamma \vdash \mu f : \tau_1 \to \tau_2 \lambda x : \tau_1 . e \Rightarrow \mu f : \tau_1 \to \tau_2 \lambda x . e_{cast} : \tau_1 \to \tau_3}$$

7.4 Exemple

En 5.4, nous avons présenté un scénario de compilation (fig. 7.1). Nous allons maintenant décrire ce processus à l'aide de SCOL et de son système de types. Nous avons déjà réalisé une version non typée de ce programme en 6.4.

7.4.1 Propriétés du programme

Nous allons définir un ensemble de propriétés qui seront définies sur le programme cible et modifiées par les passes :

— CGPROT est une propriété qui indique si le graphe d'appel de la fonction est protégé.
— COPQ est une propriété qui indique si les constantes sont opaques dans la fonction
— CMBA est une propriété qui indique si les constantes sont protégées par des techniques de MBA.
— OPT indique si la fonction a été optimisée.

On considérera que le programme sur lequel on exécute le script a été optimisé avant, mais aucune protection n'a été appliquée.

7.4.2 Type des passes

Nous reprenons les définitions de passes que nous avons réalisées en 6.4 pour y ajouter les informations de types :

$$
\begin{array}{l}
\texttt{script}(x : \langle \textsc{Opt} \ : \ \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} \ : \ \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} \ : \ \top; \partial\,\mathsf{Abs}\rangle) = \\
\quad \texttt{let\_pass}\, cgf : \ \alpha \to (\beta, \texttt{int list}, \langle \partial\,\mathsf{Abs}\rangle)\, \texttt{with}\, \beta = \alpha \oplus \textsc{CGProt} \\
\qquad \wedge \alpha <: \langle \textsc{Opt} \ : \ \mathsf{Pre}\rangle\ \texttt{from}\ \textsc{Obf.CGF}\ \texttt{in} \\
\quad \texttt{let\_pass}\, opqcst : (\texttt{int list}, \alpha) \to \beta\, \texttt{with}\, \beta = \alpha \oplus \textsc{COpq}\ \texttt{from}\ \textsc{Obf.OpqCst}\ \texttt{in} \\
\quad \texttt{let\_pass}\, mba : \ \alpha \to \beta\, \texttt{with}\, \beta = \alpha \oplus \textsc{CMBA}\ \texttt{from}\ \textsc{Obf.MBA}\ \texttt{in} \\
\quad \ldots
\end{array}
$$

Script d'obfuscation

Le script d'obfuscation en SCOL_{cast} est très similaire au script que nous avons décrit en 6.4 en ajoutant simplement les informations de types.

Le script est donc le suivant :

$$
\begin{array}{l}
\texttt{script}(x : \langle \textsc{Opt} \ : \ \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} \ : \ \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} \ : \ \top; \partial\,\mathsf{Abs}\rangle) = \\
\quad \texttt{let\_pass}\, cgf : \ \alpha \to (\beta, \texttt{int list}, \langle \partial\,\mathsf{Abs}\rangle)\, \texttt{with}\, \beta = \alpha \oplus \textsc{CGProt} \\
\qquad \wedge \alpha <: \langle \textsc{Opt} \ : \ \mathsf{Pre}\rangle\ \texttt{from}\ \textsc{Obf.CGF}\ \texttt{in} \\
\quad \texttt{let\_pass}\, opqcst : (\texttt{int list}, \alpha) \to \beta\, \texttt{with}\, \beta = \alpha \oplus \textsc{COpq}\ \texttt{from}\ \textsc{Obf.OpqCst}\ \texttt{in} \\
\quad \texttt{let\_pass}\, mba : \ \alpha \to \beta\, \texttt{with}\, \beta = \alpha \oplus \textsc{CMBA}\ \texttt{from}\ \textsc{Obf.MBA}\ \texttt{in} \\
\quad \texttt{let}\, foobar = \pi_3\, x\, \texttt{in} \\
\quad \texttt{let}\, result = cgf\ @\ foobar\, \texttt{in} \\
\quad \texttt{let}\, foobar2 = \pi_1\, result\, \texttt{in} \\
\quad \texttt{let}\, dispatch = \pi_2\, result\, \texttt{in} \\
\quad \texttt{let}\, const = \pi_3\, result\, \texttt{in} \\
\quad \texttt{let}\, foobar3 = opqcst\ @\ (const, foobar2)\, \texttt{in} \\
\quad \texttt{let}\, dispatch2 = mba\ @\ (opqcst\ @\ (const, dispatch))\, \texttt{in} \\
\qquad (\pi_1\, x, \pi_2\, x, foobar3, dispatch2)
\end{array}
$$

7.4.3 Type et insertion de *cast*

Type

En appliquant les règles de typage que nous avons décrites dans ce chapitre on peut obtenir le type de ce script d'obfuscation :

$$\langle \textsc{Opt} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} : \top; \partial\,\mathsf{Abs}\rangle \rightarrow$$
$$\langle \textsc{Opt} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} : \mathsf{Pre}; \textsc{CGProt} : \mathsf{Pre}; \textsc{COpq} :$$
$$\mathsf{Pre}; \textsc{CMBA} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle * \langle \textsc{Opt} : \mathsf{Abs}; \textsc{COpq} : \mathsf{Pre}; \partial\,\mathsf{Abs}\rangle$$

Ce type permet donc de déterminer que la fonction `foobar` résultante aura été protégée par les 3 transformations que nous avons utilisées, tandis que la fonction `dispatch` est protégée par les constantes opaques, on peut identifier que le graphe d'appels de `dispatch` ne sera pas protégé et qu'elle n'a pas non plus été optimisée.

Insertion des casts

Dans ce script, il y a un seul *cast* inséré. Ce cast est inséré dans le terme suivant :

$$\texttt{let}\ \mathit{result} = \mathit{cgf}\ @\ (< \alpha\ \texttt{with}\ \beta = \alpha \oplus \textsc{CGProt} \wedge \alpha <: \langle \textsc{Opt} : \mathsf{Pre}\,;\, \partial\top\rangle >\ \mathit{foobar})\ \texttt{in}\ \ldots$$

En effet, l'application de la passe *cgf* nécessite la présence de la propriété Opt qui est inconnu dans le terme *foobar*. D'après la règle, App4 un cast est inséré au moment de l'application. Le reste du script est identique au script d'entrée (à l'exception des informations de type qui sont retirées des termes pour correspondre à la syntaxe de SCOL_{cast}).

7.5 Conclusion

Pour que notre langage offre les possibilités de sûreté décrites dans le chapitre 5 et en 7.2.5, nous avons défini un système de types pour SCOL. Pour cela nous avons commencé par définir le type des passes, qui sont typées comme des abstractions, car les passes peuvent être vues comme des abstractions définies dynamiquement. Les passes peuvent également être définies avec un système de contraintes qui permettent, notamment, de

créer des passes dont le type de sortie dépend du type de l'entrée en gérant les conditions d'application de ces passes. Nous avons également défini un type pour les programmes (que nous avons pour le moment limité aux fonctions), qui utilise une rangée pour contenir les propriétés correspondantes aux informations connues sur le programme.

Le langage utilise le typage graduel, car certains termes du programme sont intrinsèquement dynamiques. En particulier, le programme sur lequel le script d'obfuscation est appliqué est dynamique. Le système de types insère des *cast* dans le script, lorsqu'une vérification de type est nécessaire. Il s'agit de la première étape de la compilation, après cette étape le script est évalué statiquement comme nous l'avons décrit dans le chapitre 6. Puis, une fois évalué statiquement, le script est traduit vers du code C++, par le processus décrit dans le chapitre 8, qui sera compilé et exécuté par le gestionnaire de passes de LLVM.

Chapitre 8

Gestionnaire de passes et exécution de programme SCOL

8.1 Introduction

Dans les chapitres précédents, nous avons présenté les premières étapes de la compilation d'un script d'obfuscation SCOL : la vérification de type, l'insertion de *cast* pour la vérification dynamique de type et une évaluation statique de certains termes du script. Comme indiqué dans la figure 5.6, le code du script à l'entrée de ce chapitre est dans le langage SCOL_{dyn}. Ce chapitre décrit l'intégration de SCOL dans LLVM. Cela va notamment nécessiter la génération de code C++, et donc notamment, la traduction de notre approche fonctionnelle de la gestion des passes, et la traduction du système de types pour générer du code bien typé au sens du C++.

Ce chapitre est décomposé en quatre parties. Dans un premier temps, nous allons présenter le fonctionnement technique de notre gestionnaire de passes et les différents outils qui le composent. Dans un second temps, nous allons présenter la traduction du système de types et le fonctionnement de la vérification dynamique de type. Ensuite, nous présenterons les règles de traduction qui génèrent le code C++ du script d'obfuscation, qui sera compilé par un compilateur C++ traditionnel [1] et exécuté par le gestionnaire de passes. Enfin, nous présenterons également un exemple pour montrer ce fonctionnement dans un cas concret.

Ce qui est présenté dans ce chapitre est un démonstrateur, destiné à montrer les possibilités de notre langage de description de script d'obfuscation. Ce démonstrateur n'a donc pas vocation à être utilisé industriellement, car il possède plusieurs limitations, mais pourra être amélioré dans le futur.

1. En l'occurrence, nous utiliserons CLang, mais le code n'utilise pas de comportements spécifiques à ce compilateur et fonctionne également avec GCC dans nos tests.

8.2 Gestionnaire de passes

8.2.1 Introduction

Dans cette section, nous allons décrire comment les différentes entités et constructions de SCOL sont représentées dans le code C++ à l'intérieur de notre gestionnaire de passes LLVM. Nous allons présenter la forme que prennent les programmes, les passes et les propriétés dans LLVM. Nous allons également présenter le principe de fonctionnement de notre gestionnaire de passes.

8.2.2 Programmes

Dans LLVM, les programmes sont décrits grâce à plusieurs classes spécifiques à chaque structure : `Module`, `Function`, `Loop`, `Region` ... Dans SCOL, nous nous sommes limités aux fonctions, nous avons donc seulement besoin de représentation pour les fonctions. Nous définissons une classe `SCOLFunction` qui hérite de `Function`, ce qui permet d'utiliser toutes les méthodes utiles pour LLVM et d'être compatible avec les passes existantes. Dans cette sous-classe, nous ajoutons principalement un attribut pour gérer les propriétés. Les propriétés sont enregistrées sous la forme d'un dictionnaire associant le nom de la propriété à son état Pre, Abs ou $\top$. La classe `SCOLFunction` contient également des méthodes pour manipuler les propriétés : ajouter/modifier la valeur d'une propriété, récupérer la valeur d'une propriété ou récupérer l'ensemble des propriétés (c'est-à-dire le type de la fonction).

8.2.3 Passes

Implémentation usuelle des passes dans LLVM

Les passes de LLVM doivent suivre une spécification particulière dans leurs implémentations. Toutes les passes doivent être des sous-classes de la classe `Pass` et implémenter certaines méthodes virtuelles héritées de `Pass`. En général, les passes n'hériteront pas directement de `Pass`, mais plutôt d'une des sous-classes suivantes :

— `ModulePass`. Il s'agit de la sous-classe la plus générique. Elle indique que la passe manipule le programme entier, c'est-à-dire qu'une passe peut ajouter ou supprimer des fonctions ou modifier un nombre quelconque de fonctions dans un ordre non défini. Cette sous-classe donne donc peu d'information sur les effets de la passe au gestionnaire de passes qui ne peut donc faire que peu d'optimisation sur son

exécution.

— `CallGraphSCCPass`. Il s'agit de la sous-classe utilisée par les passes qui ont besoin de traverser le graphe d'appel. Elles peuvent modifier le graphe d'appels, mais elles doivent conserver la cohérence de la représentation du graphe, c'est-à-dire que c'est à la passe de s'assurer que la représentation du graphe d'appel dans LLVM correspond au programme après l'exécution de la passe, elle peut modifier le graphe pour y arriver. Il y a également certaines restrictions supplémentaires sur les possibilités des passes. Par exemple, elles ne peuvent pas ajouter ou retirer de fonctions après l'initialisation.

— `FunctionPass`. Chaque application de la passe modifie une seule et unique fonction. De plus, l'application de cette passe sur une fonction doit être indépendante de son application sur les autres fonctions. Dans le gestionnaire de passes usuel, ces passes sont appliquées à chacune des fonctions du programme.

— `LoopPass`. De façon similaire aux `FunctionPass`, les `LoopPass` s'appliquent sur chaque boucle indépendamment d'autres boucles dans la fonction et le programme.

— `RegionPass`. Elles ont un fonctionnement similaire aux `LoopPass`, mais s'appliquent sur les régions délimitées par une entrée et une sortie (c'est-à-dire un sous graphe du graphe de flot de contrôle) plutôt que sur les boucles.

Les descriptions données ici sont un aperçu et sont incomplètes. Chacune de ces sous-classes a des spécificités et restrictions supplémentaires qui sont décrites dans la documentation de LLVM [199].

Ces sous-classes, excepté pour `ModulePass` que nous détaillerons ensuite, fonctionnent de façon similaire dans le gestionnaire de passes de LLVM. Pour écrire une passe, il est en général nécessaire d'implémenter plusieurs méthodes abstraites :

— `doInitialization`

— `doFinalization`

— `runOnX` où `X` est la structure correspondante à la sous-classe (`Function`, `Module`, `Loop`...).

Les méthodes `doInitialization` et `doFinalization` s'exécutent respectivement avant et après l'exécution de la passe, et permettent de faire certaines initialisations (resp. finilisations) et ont également des possibilités de modifications supplémentaires par rapport au cœur de la passe (par exemple, `doInitialization` dans une `FunctionPass` peut ajouter et supprimer des fonctions, contrairement à `runOnFunction`). La méthode `runOnX` prend un argument, qui est l'instance correspondante à la structure (`Function`, `Loop`,...) et c'est

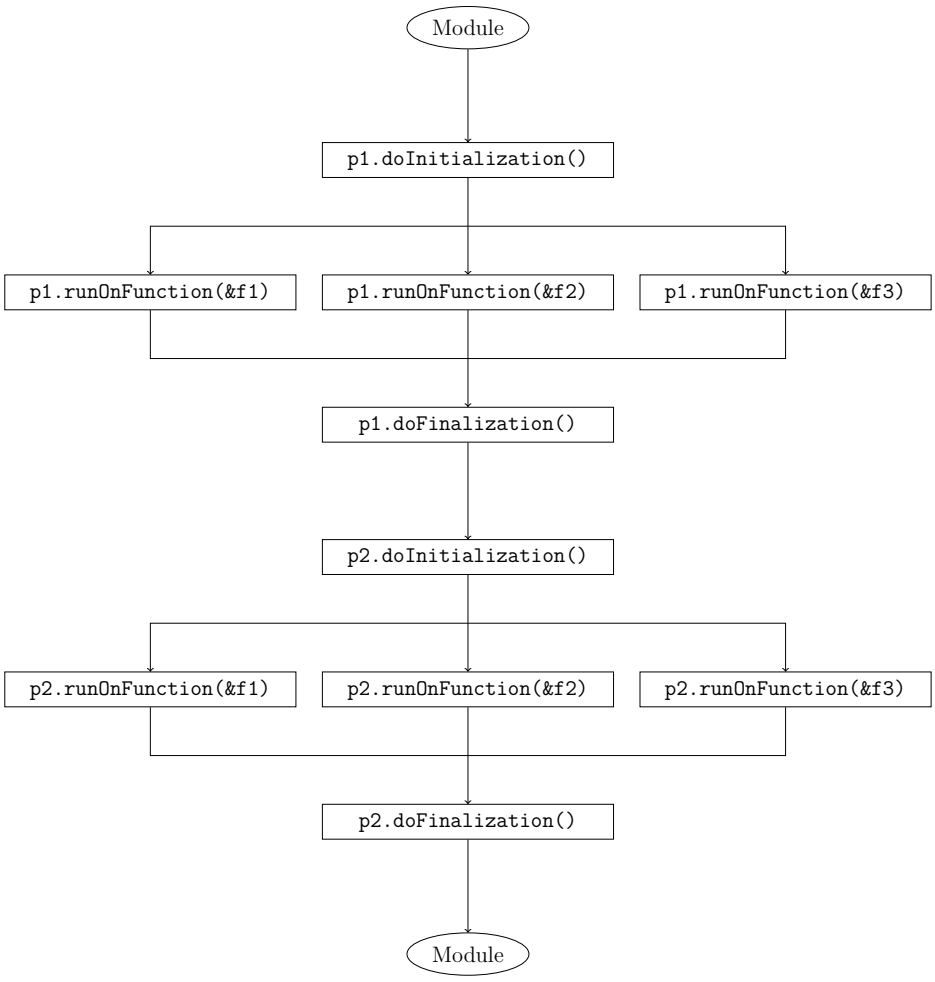

FIGURE 8.1 – Processus d'exécution de deux `FunctionPass` (`p1` et `p2`) sur un module contenant trois fonctions (`f1`, `f2` et `f3`) par le gestionnaire de passes de LLVM

cette méthode qui effectue la transformation de la structure.

Le gestionnaire de passes exécute la méthode `doInitialization` une fois, puis la méthode `runOnX` sur chaque instance de la structure. Grâce aux limitations possibles dans les transformations possibles dans la méthode `runOnX`, ces exécutions sont indépendantes et peuvent donc potentiellement être exécutées en parallèle. Après l'exécution de `runOnX` sur toutes les instances du programme, le gestionnaire de passes exécute la méthode `doFinalization`. La figure 8.1 montre un exemple de ce processus d'exécution avec deux `FunctionPass` appliquées sur un module contenant 3 fonctions.

La sous-classe `ModulePass`, ne contient que la méthode, `runOnModule` car la chaîne de transformation ne s'applique que sur un seul module. Il n'y a donc qu'une seule applica-

tion de la méthode `runOnModule` de la passe et donc les méthodes d'initialisation et de finalisation ne sont pas nécessaires.

Chacune de ces méthodes retourne un booléen. Ce booléen indique si la passe modifie la structure sur laquelle, elle s'exécute. Cela permet notamment au gestionnaire de passes de gérer le cache d'analyse (cf. 8.2.3).

Interactions entre les passes dans LLVM

Le gestionnaire de passes de LLVM permet certaines interactions entre les passes, en particulier dans la gestion des analyses. Certaines passes ont besoin d'informations sur le programme sur lequel elles s'exécutent et ont donc besoin d'analyser le programme. Pour éviter la répétition de code et optimiser l'exécution de la chaîne de compilation, ces analyses sont en général implémentées dans des passes spécifiques plutôt qu'à l'intérieur des passes de transformation.

En implémentant la méthode `getAnalysisUsage` dans la passe, LLVM permet de décrire les interactions de la passe avec les passes d'analyse. En particulier, cela permet :

— d'indiquer les passes d'analyse requises par la passe
— d'indiquer les passes d'analyse qui sont préservées et celles qui ne le sont pas
— de récupérer les informations contenues dans les analyses

Par défaut, si cette méthode n'est pas implémentée, le gestionnaire de passes considère que la passe ne requiert aucune analyse, et les invalide toutes.

Dans LLVM, c'est donc le gestionnaire de passes qui se charge de planifier l'exécution des analyses en fonction des besoins des passes.

Définition et contenu des passes SCOL

Dans SCOL, nous avons choisi d'offrir plus de liberté sur certains points dans la définition des passes (cf. 5.2). En particulier, les entrées et sorties des passes ne sont pas fixées. Dans SCOL, une passe est donc une fonction avec une signature quelconque.

Il y a cependant quelques restrictions sur l'implémentation des passes SCOL. Afin de pouvoir être manipulés par le langage de script, les paramètres d'entrée et sortie de la passe doivent être des entités représentables dans le langage de script (des fonctions, des passes, des programmes, des entiers, des booléens, des listes, des tuples...). De plus, comme nous l'avons décrit en 5.2.3, les passes ne doivent pas avoir d'effet de bord, c'est-à-dire en particulier, qu'elles ne doivent pas modifier directement le programme, mais doivent retourner une version modifiée du programme.

Contrairement aux passes usuelles de LLVM, les passes SCOL n'ont pas à indiquer les informations sur les analyses qu'elles affectent et dont elles ont besoin, ceci étant géré directement dans le script d'obfuscation par le système de types.

Propriétés

Comme nous l'avons indiqué en 8.2.2, les valeurs des propriétés associées aux fonctions sont stockées dans la classe `SCOLFunction`, sous la forme d'un dictionnaire associant le nom de la propriété à sa valeur. Les passes doivent implémenter la mise à jour des propriétés des fonctions qu'elles manipulent lorsqu'elles les modifient. Au départ de la chaîne de compilation, les propriétés de toutes les fonctions sont initialisées à Abs. Si l'utilisateur souhaite spécifier une présence initiale différente pour les propriétés d'une fonction spécifique, cela peut être réalisé à l'aide d'annotation de fonction dans le code source du programme.

Utilisation des passes existantes dans SCOL

De nombreuses passes d'optimisation, mais également d'obfuscation, ont déjà été développées pour le gestionnaire de passes traditionnel de LLVM. Ces passes ne sont pas utilisables directement avec SCOL, mais il peut être intéressant de les réutiliser pour ne pas les réimplémenter complètement. Nous allons présenter quelques principes pour convertir les passes usuelles vers des passes SCOL. Comme nous nous sommes limités à des passes manipulant des fonctions dans SCOL, nous allons discuter ici de la conversion des `FunctionPass` [2].

Si la passe n'utilise pas les méthodes `doInitialization` et `doFinalization`, il est possible de créer une fonction pour exécuter la passe. La fonction aura la forme suivante :

```
SCOLFunction * passSCOL (SCOLFunction * foo){
  SCOLFunction * copy = foo.clone();
  pass.runOnFunction(copy);
  updateProperties(copy);
  return copy;
}
```

2. Il est également possible de convertir les autres types de passes, mais cela demandera plus de changement dans l'implémentation des passes.

Cela permet de réutiliser entièrement le code de la transformation sans le modifier. Le clonage de la fonction permet de transformer la passe en passe « fonctionnelle », c'est-à-dire retirer la modification en place. Ce clonage assure la transformation fonctionnelle si les contraintes de création de `FunctionPass` de LLVM sont respectées. La fonction `updateProperties` doit être ajoutée et appelée pour gérer les propriétés associées à la fonction (cf. 8.2.3). Ce processus pourrait être partiellement automatisé pour certaines passes[3], il sera tout de même nécessaire d'écrire le contenu de la fonction `updateProperties` associée.

Si la passe utilise les méthodes `doInitialization` et `doFinalization`, il faut réécrire le code de ces méthodes pour prendre en compte les restrictions du langage de script (en particulier, il ne faut pas d'effet de bord dans ces méthodes).

8.2.4 Exécution du processus de compilation

Pour notre implémentation, nous avons choisi d'intégrer le script d'obfuscation à l'intérieur d'une `ModulePass` appelée par le gestionnaire de passe par défaut de LLVM. Ce choix permet de faciliter l'implémentation de notre démonstrateur pour plusieurs raisons. Cela permet de l'intégrer au gestionnaire de passes existant de LLVM et facilite donc l'implémentation du script d'obfuscation dans CLang. Cette implémentation permet aussi de réutiliser une partie des passes d'optimisation et d'analyse incluses dans LLVM sans aucune modification sur ces passes. Cela permet donc de réaliser des scripts d'obfuscation partiels intégrés à une chaîne d'optimisation standard ce qui rend la création de scripts de démonstration plus rapide et plus simple. Cela n'empêche pas de créer des scripts de compilation complets, en modifiant le gestionnaire de passes pour que seule notre `ModulePass` (qui exécute notre script d'obfuscation complet) soit appliquée. La limitation principale de cette implémentation est qu'elle n'est pas modulaire et que le script d'obfuscation est directement intégré au compilateur CLang.

Cette `ModulePass` transmet au script SCOL le programme sous la forme de la liste (ou tuple) de ses fonctions et les autres arguments potentiels du script. Il attend en retour une liste de ces fonctions, potentiellement transformées par l'application de passes. Notre implémentation restreint l'ordre des paramètres d'entrée : la liste de fonctions doit être le premier argument, et le résultat du script : une liste de fonctions. Cela implique quelques restrictions supplémentaires par rapport à la définition de SCOL, mais ne limite pas

3. Celles qui n'utilisent pas `doInitialization` et `doFinalization` et qui respectent les contraintes décrites dans la référence de LLVM décrites en 8.2.3.

réellement les possibilités dans l'écriture de script d'obfuscation. Cependant, cela rend impossible la composition de plusieurs scripts d'obfuscation partiels pour composer le script complet.

8.3 Types et *cast*

8.3.1 Introduction

L'implémentation dans LLVM de SCOL doit gérer deux systèmes de types différents : le code C++ doit être bien typé au sens du C++ pour que le code se compile et s'exécute correctement, et les types SCOL doivent être vérifiés dynamiquement par des *casts*. Nous aurons donc besoin de traduire le type SCOL de deux façons différentes, dans le système de types de C++ et dans un système de types équivalent à celui de SCOL. Dans cette section, nous allons présenter ces deux systèmes de types et règles de traduction, ainsi que l'implémentation de la vérification dynamique de type.

8.3.2 Traduction des types pour le typage C++

Le code généré doit être bien typé au sens du C++, donc certaines informations de types doivent également être traduites en C++ pour générer du code bien typé. En particulier, pour les fonctions récursives, nous utilisons les informations de types pour la définition de ces fonctions en C++. Ces fonctions doivent être typées explicitement et doivent être bien typées au sens du langage C++. Cette traduction sera notée : $\tau \rightsquigarrow \tau_{cpp}$ où τ est le type SCOL et τ_{cpp} est la traduction en C++.

Les types SCOL et les types C++ utilisés pour les scripts SCOL sont listés ci-dessous :

$$
\begin{aligned}
\tau \quad &::= \tau \to \tau \mid \alpha \mid \tau \,\texttt{with}\, C_{main} \mid \\
&\quad < row > \mid \texttt{int} \mid \texttt{bool} \mid \tau\texttt{list} \mid \tau * ... * \tau \\
\tau_{cpp} &::= \texttt{std} :: \texttt{function} < \tau_{cpp}(\tau_{cpp}) > \mid \texttt{SCOLFunction} \mid \\
&\quad \texttt{int} \mid \texttt{bool} \mid \texttt{SCOLList} < \tau_{cpp} > \mid \texttt{std} :: \texttt{tuple} < \tau_{cpp}, ..., \tau_{cpp} >
\end{aligned}
$$

Nous utilisons `std :: function` pour décrire le type des abstractions et des passes et `SCOLFunction` pour le type des fonctions (du programme cible). Notons que pour les types C++, nous n'avons pas besoin de rentrer dans les détails des rangées et des contraintes pour cette traduction, car nous utilisons ce type uniquement pour que le code C++ soit

bien typé, et non pas pour les vérifications dynamiques de type SCOL, qui sont décrites en 8.3.4.

Les règles sont maintenant assez directes. D'abord la traduction des entiers (TInt), booléens (TBool), listes (TList), tuples (TTuple) et abstractions(TAbs) sont une traduction directe vers les types équivalents en C++.

$$\textsc{TInt}\ \texttt{int} \rightsquigarrow \texttt{int} \qquad\qquad \textsc{TBool}\ \texttt{bool} \rightsquigarrow \texttt{bool}$$

$$\textsc{TList}\ \frac{\tau \rightsquigarrow \tau_{cpp}}{\tau\texttt{list} \rightsquigarrow \texttt{SCOLList} < \tau_{cpp} >}$$

$$\textsc{TTuple}\ \frac{\forall i \in [1,n] \tau^i \rightsquigarrow \tau^i_{cpp}}{\tau^1 * ... * \tau^n \rightsquigarrow \texttt{std} :: \texttt{tuple} < \tau^1_{cpp}, ..., \tau^n_{cpp} >}$$

$$\textsc{TAbs}\ \frac{\tau \rightsquigarrow \tau_{cpp} \qquad \tau' \rightsquigarrow {\tau_{cpp}}'}{\tau \to \tau' \rightsquigarrow \texttt{std} :: \texttt{function} < {\tau_{cpp}}'(\tau_{cpp}) >}$$

Plusieurs types SCOL représentent des fonctions du programme cible. En effet, les rangées (TRow) et les variables de type (TVar) ne peuvent désigner que des fonctions du programme. Les règles sont les suivantes :

$$\textsc{TRow} < row > \rightsquigarrow \texttt{SCOLFunction} \qquad\qquad \textsc{TVar}\ \alpha \rightsquigarrow \texttt{SCOLFunction}$$

Enfin, le type contraint suit les mêmes règles de traduction que le type sans la contrainte, car la contrainte n'apporte que des informations sur les propriétés qui ne sont pas utilisées dans le type C++. La règle TConstraint est la suivante :

$$\textsc{TConstraint}\ \frac{\tau \rightsquigarrow \tau_{cpp}}{\tau\ \texttt{with}\ C_{main} \rightsquigarrow \tau_{cpp}}$$

8.3.3 Implémentation du *cast*

Notre gestionnaire de passes, en plus des autres concepts que nous avons présentés en 8.2, doit également gérer l'exécution des *cast*. Nous rappelons que les *casts* sont insérés lors de la compilation du script d'obfuscation. Ils sont insérés lorsque le script nécessite que le terme soit d'un type particulier, mais que ce type ne peut pas être vérifié statiquement. C'est donc le rôle du gestionnaire de passes de vérifier que le type réel à l'exécution est compatible avec le type dans lequel le terme est *casté*.

Pour cela, nous avons implémenté une fonction pour réaliser le *cast*. Cette fonction prend deux arguments : le type destination du *cast* et le terme à *caster*. Cette fonction renvoie le terme passé en argument lorsque le *cast* est possible. Si le *cast* n'est pas possible, car le type destination et le type réel ne sont pas compatibles, alors une exception est levée.

Dans le cœur de la fonction, la vérification peut être divisée selon trois cas :

— le type SCOL a une correspondance directe avec le type C++ (entier, booléen, liste ...)
— le type utilise les propriétés
— le type est défini par des contraintes

Nous allons maintenant présenter rapidement le principe du *cast* pour chacun de ces cas.

Le type SCOL a une correspondance directe avec le type C++

Dans ce cas, on peut vérifier si le *cast* est possible directement avec les types C++. Cela est trivial pour les entiers et les booléens. Pour les listes et les tuples, ce cas permet de vérifier qu'il s'agit bien de la bonne structure de données, mais il faut également vérifier (récursivement) si les éléments contenus dans cette structure sont également compatibles entre le type destination et le type réel.

Le type utilise les propriétés

Lorsque le type utilise des propriétés, pour vérifier la compatibilité des types, il est nécessaire de vérifier que chaque propriété possède une présence compatible. Si une propriété a une présence identique dans les deux types, alors elles sont compatibles. Si la présence de la propriété du type de destination est $\top$ alors elle est compatible quelle que soit la présence de la propriété dans le type réel. Enfin, si la présence du type réel est,

$\top$ mais pas celle associée au type de destination, alors nous avons plusieurs possibilités. D'abord, ce cas de figure signifie que l'on ne sait pas quelle est la présence réelle associée à cette propriété dans le programme, c'est-à-dire qu'aucune passe ou analyse précédente n'a calculé la présence de cette propriété. Les différentes possibilités pour gérer ce cas de figure sont les suivantes :

— On considère que le *cast* est toujours possible. Dans ce cas, il n'y aura pas d'exécution correcte rejetée, mais certaines exécutions incorrectes ne seront pas détectées. Il est possible de prévenir l'utilisateur avec un message (par exemple un *warning* indiquant que ce *cast* a été supposé possible et que cela a pu générer des erreurs).
— On considère que la présence par défaut pour le *cast* est Abs, c'est-à-dire que le *cast* ne sera pas valide pour les propriétés qui requièrent Pre. Autrement dit, les propriétés sont considérées comme absentes si elles n'ont pas été explicitement indiquées comme présentes. Inversement, on peut considérer que la présence par défaut pour le *cast* est Pre. Autrement dit, les propriétés sont considérées comme présentes si elles n'ont pas été explicitement indiquées comme absentes. Selon le choix des propriétés, l'un ou l'autre de ces choix peut être plus intuitif. Ils peuvent cependant générer des erreurs.
— On calcule la présence de la propriété. Pour cela, il est nécessaire d'avoir une fonction qui est capable de calculer la présence d'une propriété d'une fonction en connaissant complètement cette fonction. Idéalement, nous aimerions utiliser cette possibilité, car elle évite les faux positifs et les faux négatifs, mais nous avons permis de définir des propriétés qui ne peuvent pas être calculées. Dans le futur, on pourrait imaginer permettre d'associer une fonction à certaines propriétés, qui calcule la présence associée à cette propriété lorsqu'elle est inconnue (cf. 9.2). Si aucune fonction de calcul n'est définie, on pourra continuer à utiliser une des autres possibilités.
— On considère que le cast est impossible. Cela empêchera certaines exécutions correctes de script. Par contre, aucune exécution incorrecte n'est possible. Il s'agit de la seule solution qui conserve la correction apportée par le système de types.

Nous avons fait le choix de considérer que le *cast* est impossible dans ce cas. Cela permet d'assurer la sûreté du script d'obfuscation. Cependant, cela oblige le développeur à s'assurer que la présence de toutes les propriétés nécessaires est connue, via les annotations dans le code ou l'application préalable de passes (d'analyse particulièrement).

Le type est défini par les contraintes

Dans ce cas, la fonction de *cast* va vérifier que le type réel satisfait les contraintes, c'est-à-dire si le type réel est une des solutions des contraintes. S'il y a d'autres variables de type dans les contraintes, nous pouvons considérer n'importe quelle valeur pour ces variables, nous recherchons s'il existe une valeur de cette variable pour laquelle le type réel est une solution. Nous n'avons pas besoin de choisir une solution pour les autres variables, si le type de ces variables est nécessaire pour l'exécution du script, un autre *cast* aura été inséré dans le programme. Ce *cast* sera possiblement vérifié plus loin dans l'exécution du script.

Par exemple, le type $\langle \text{CFProt} : \mathsf{Abs}; \text{AProp} : \mathsf{Pre}; \partial\top \rangle$ est une solution pour α de la contrainte $\alpha\,\texttt{with}\,((\alpha <: \langle \text{CFProt} : \mathsf{Abs}; \partial\top\rangle) \wedge \beta = \alpha \oplus \text{CFProt})$.

8.3.4 Traduction du type dans les *casts*

Le *cast* vérifie le type au sens de SCOL, nous devons donc utiliser une équivalence stricte avec ceux-ci. Cependant, ces informations de type n'ont pas nécessairement besoin d'être considérées comme des types au sens du langage C++. Ces types seront donc modélisés par des objets en C++. Chaque classe représente une construction de type de SCOL, avec une traduction directe entre les deux. Chaque classe de type héritera de la classe `SCOLType`.

Nous ne décrirons pas ici explicitement toutes les traductions existantes, car elles sont toutes construites de la même manière, mais l'ensemble sera décrit en annexe E. Nous présentons ici un extrait afin de présenter le principe :

$$
\begin{aligned}
\tau \quad & ::= \tau \to \tau \mid \tau\ \texttt{with}\ C_{main} \mid < row > \mid \texttt{int} \mid ... \\
\mathit{SCOLType} \ & ::= \texttt{SCOLTypeFunction}(\mathit{SCOLType}, \mathit{SCOLType}) \mid \\
& \quad\ \texttt{SCOLTypeConstraint}(\mathit{SCOLType}, \texttt{SCOLConstraint}) \mid \\
& \quad\ \texttt{SCOLTypeRow}(\texttt{SCOLRow}) \mid \texttt{SCOLTypeInt}
\end{aligned}
$$

Nous notons la traduction entre ces types sous la forme suivante : $\tau \hookrightarrow \mathit{SCOLType}$. Vu qu'il y a une traduction évidente entre les types SCOL et C++, les règles de traduction peuvent être obtenues directement. Par exemple, la règle pour la traduction des types d'abstractions, TCAbs est décrite ci-dessous. Les autres règles auront la même forme.

$$\textsc{TCAbs}\ \frac{\tau \hookrightarrow \mathit{SCOLType} \qquad \tau' \hookrightarrow \mathit{SCOLType}'}{\tau \to \tau' \hookrightarrow \texttt{SCOLTypeFunction}(\mathit{SCOLType}, \mathit{SCOLType}')}$$

8.4 Traduction du script SCOL

8.4.1 Introduction

Maintenant que nous avons décrit le principe global de l'exécution des scripts d'obfuscation, nous allons décrire la dernière étape de traduction du script. Cette traduction génère du code C++ intégré à la `ModulePass` que nous avons décrite en 8.2.4. Il s'agit de la dernière étape de traduction du script avant son exécution tel que décrit dans la figure 5.6.

8.4.2 Traduction de SCOL minimal

Règles de traduction

Commençons par décrire les règles de traduction pour la version minimale de SCOL. Rappelons d'abord la syntaxe des termes de SCOL_{dyn} qui sont les constructions que nous allons traduire en C++ :

$$\begin{aligned} e_{dyn} \quad &::= \texttt{RunPass}(\text{PASS}, e_{dyn}) \mid e_{dyn}\ @_{dyn}\ e_{dyn} \mid x \mid v \\ \mathbb{V} :: v \ &::= \lambda_{dyn} x.e_{dyn} \mid \texttt{pass}(\text{PASS}) \end{aligned}$$

Cette étape de traduction est décrite par des règles de la forme $e_{dyn} \rightsquigarrow e_{cpp}$, où e_{dyn} est le terme dans SCOL_{dyn} est e_{cpp} le terme traduit en C++. Commençons par décrire la règle de traduction pour l'application des passes. La règle CRunPass applique simplement la passe (qui est une fonction existante dans l'environnement C++) à son paramètre.

$$\textsc{CRunPass}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{RunPass}(\text{PASS}, e_{dyn}) \rightsquigarrow \text{PASS}(e_{cpp})}$$

De même, la règle CPass indique que la passe est traduite par la fonction correspondante et CVar indique que la variable est remplacée par la variable correspondante :

$$\textsc{CPass}\ \texttt{pass}(\text{PASS}) \rightsquigarrow \text{PASS} \qquad\qquad \textsc{CVar}\ x \rightsquigarrow x$$

La règle CApp indique que la traduction de l'application est un appel de fonction (terme de gauche) à un argument (terme de droite). La fonction peut être une fonction nommée, une fonction lambda C++ non nommée ou le retour d'une fonction qui sera une référence d'une fonction [4]. La règle sera la même dans les trois cas.

$$\textsc{CApp}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn} @_{dyn} e^2_{dyn} \rightsquigarrow e^1_{cpp}(e^2_{cpp})}$$

Enfin pour la traduction de l'abstraction, nous utilisons la règle CAbs qui traduit le corps de l'abstraction et l'intègre au corps d'une fonction lambda C++. Notons que les fonctions lambda que nous utilisons ne peuvent être définies qu'à partir de C++14, car nous utilisons des fonctions lambda génériques.

$$\textsc{CAbs}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\lambda_{dyn} x.e_{dyn} \rightsquigarrow [=](\texttt{const auto\& x})\{\texttt{return}\ e_{cpp};\}}$$

Notons que nous utilisons les fonctions lambda de C++. Elles sont de la forme `[ captures ] ( params ) -> ret { body }`. L'expression `[ captures ]` indique quelles variables de l'environnement sont capturées dans l'expression. Nous utiliserons toujours `[=]` qui indique que toutes les variables de l'environnement sont capturées par valeurs. Cela permet de reproduire le comportement des λ-expressions fonctionnelles. `( params )` indique les paramètres d'entrée de la fonction. `-> ret` est optionnel [5] et indique le type de retour de la fonction. Enfin, `{ body }` contient le corps de la fonction lambda.

Exemple

Prenons un script SCOL simple dont nous avons décrit les premières étapes de traduction en 6.2.3 [6]. Ce script se traduit dans SCOL_{dyn} comme suit :

4. Dans ce cas, le compilateur déréférence automatiquement la référence à la fonction.
5. Si le compilateur est capable de déterminer le type de retour, ce qui est le cas ici.
6. Nous avons simplement renommé les passes ici.

$$
\begin{array}{l}
\lambda_{dyn}x.(\\
\quad \texttt{RunPass}(\textsc{Opt}, \\
\quad\quad \texttt{RunPass}(\text{CFGF}, \\
\quad\quad\quad \texttt{RunPass}(\text{BCF}, x))) \\
)
\end{array}
$$

En appliquant les règles précédentes, nous obtenons facilement le code C++ suivant :

```
[=](const auto& x){
  return OPT(CFGF(BCF(x)));
}
```

On remarque que ce script est traduit en une fonction lambda. Comme nous l'avons indiqué en 8.2.4 cette fonction est ensuite appliquée sur le paramètre d'entrée par notre gestionnaire de passes.

Traduction du *cast*

Grâce aux règles de traduction de type décrites en 8.3.4, nous pouvons définir la règle CCast pour la traduction de l'opération de *cast*. Dans cette règle nous utilisons la fonction utilitaire polymorphe[7] `SCOLCast` prenant deux arguments : le type (au sens SCOL) et le terme à *cast* ; et retournant le terme. Si le *cast* est impossible, car le type cible et le type réel ne sont pas compatibles, une exception est lancée.

$$\textsc{CCast}\ \frac{\tau \hookrightarrow \texttt{SCOLType} \qquad e_{dyn} \rightsquigarrow e_{cpp}}{(<\tau>\ \ e_{dyn}) \rightsquigarrow \texttt{SCOLCast}(\texttt{SCOLType}, e_{cpp})}$$

8.4.3 Traduction des extensions

Nous nous intéressons maintenant à la traduction des différentes extensions à SCOL.

Entiers, booléens et structures conditionnelles

Nous rappelons que les constructions pour les entiers, les booléens et les structures conditionnelles ajoutées dans le langage dynamique sont les suivantes :

7. Polymorphe au sens des types C++.

$$\begin{aligned} constant_{dyn} &::= \texttt{true}_{dyn} \mid \texttt{false}_{dyn} \mid \texttt{int} \\ e_{dyn} &::= \ldots \mid \texttt{if}_{dyn}\, e_{dyn}\, \texttt{then}\, e_{dyn}\, \texttt{else}\, e_{dyn} \mid unop_{dyn}\, e_{dyn} \mid e_{dyn}\, binop_{dyn}\, e_{dyn} \end{aligned}$$

Commençons par décrire les règles triviales pour les constantes : CTrue, CFalse et CInt qui indique simplement que ces constantes sont traduites par leurs équivalentes en C++ :

$$\textsc{CTrue}\ \texttt{true} \rightsquigarrow \texttt{true} \qquad \textsc{CFalse}\ \texttt{false} \rightsquigarrow \texttt{false} \qquad \textsc{CInt}\ \texttt{int} \rightsquigarrow \texttt{int}$$

De même, pour les opérateurs nous avons les règles CUnop et CBinop, qui traduisent les opérandes et appliquent l'opérateur sur les opérandes. Nous ne précisons pas ici la traduction pour chacun des opérateurs, il s'agit bien sûr des opérateurs équivalents en C++.

$$\textsc{CUnop}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{unop_{dyn}\ e_{dyn} \rightsquigarrow unop_{cpp}\ e_{cpp}}$$

$$\textsc{CBinop}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn}\ binop_{dyn}\ e^2_{dyn} \rightsquigarrow e^1_{cpp}\ binop_{cpp}\ e^2_{cpp}}$$

Enfin, pour la structure conditionnelle, nous traduisons chacun des termes de la structure pour les insérer dans une condition ternaire en C++, comme l'indique la règle CIf :

$$\textsc{CIf}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp} \qquad e^3_{dyn} \rightsquigarrow e^3_{cpp}}{\texttt{if}_{dyn}\, e^1_{dyn}\, \texttt{then}\, e^2_{dyn}\, \texttt{else}\, e^3_{dyn} \rightsquigarrow e^1_{cpp}\ ?\ e^2_{cpp}\ :\ e^3_{cpp}}$$

Listes

Nous rappelons que les constructions pour les listes dans le langage dynamique sont les suivantes :

$$
\begin{array}{lll}
\mathit{constant}_{dyn} & ::= \ldots \mid [\,]_{dyn} \\
e_{dyn} & ::= \ldots \mid \texttt{hd}_{dyn}\, e_{dyn} \mid \texttt{tl}_{dyn}\, e_{dyn} \\
v & ::= \ldots \mid e_{dyn} ::_{dyn} e_{dyn}
\end{array}
$$

Pour utiliser les listes dans le code C++, nous avons implémenté une classe `SCOLList` qui contient des méthodes permettant d'utiliser des listes dans un style fonctionnel en C++. En effet, le C++ standard ne dispose pas de listes qui peuvent s'utiliser directement dans un style fonctionnel. Nous avons donc créé une classe utilitaire facilitant la traduction du script SCOL. Cette classe utilise `std::list` en interne et les fonctions qu'elles offrent ont une sémantique identique aux méthodes équivalentes dans les langages fonctionnels (celles décrites en 6.3.3).

La règle de traduction CEmptyList pour la liste vide est donc simplement la suivante :

$$\textsc{CEmptyList}\ [\,]_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{EmptyList}()$$

De même, les opérateurs sur les listes se traduisent trivialement en traduisant les opérandes et en utilisant notre implémentation de l'opérateur (règles CHd, CTl et CCons) :

$$\textsc{CHd}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{hd}\, e_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{hd}(e_{cpp})} \qquad \textsc{CTl}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{tl}\, e_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{tl}(e_{cpp})}$$

$$\textsc{CCons}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn} ::_{dyn} e^2_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{cons}(e^1_{cpp}, e^2_{cpp})}$$

Tuples

Commençons par rappeler les constructions qui utilisent les tuples dans SCOL_{dyn}.

$$
\begin{array}{lll}
e_{dyn} & ::= \ldots \mid \pi^i_{dyn}\, e_{dyn} \\
v & ::= \ldots \mid (e_{dyn}, \ldots, e_{dyn})_{dyn}
\end{array}
$$

Les constructions sur les tuples peuvent être traduites quasiment directement par leur équivalent en C++ (les tuples de la librairie standard). Cela nous donne les règles

TuplePROJ et TupleCons :

$$\textsc{TuplePROJ}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\pi^{i}_{dyn}\, e_{dyn} \rightsquigarrow \texttt{std} :: \texttt{get} < \texttt{i} > (e_{cpp})}$$

$$\textsc{TupleCons}\ \frac{\forall i \in [1,n], e^{i}_{dyn} \rightsquigarrow e^{i}_{cpp}}{(e^{1}_{dyn}, ..., e^{n}_{dyn})_{dyn} \rightsquigarrow \texttt{std} :: \texttt{make\_tuple}(e^{1}_{cpp}, ..., e^{2}_{cpp})}$$

Récursion

Comme pour les extensions précédentes, nous commençons par rappeler la forme des fonctions récursives dans SCOL$_{dyn}$:

$$v ::= \mu_{dyn} f : \tau \rightarrow \tau . \lambda x . e_{dyn}$$

Pour les fonctions récursives, le principe est similaire aux autres abstractions : la fonction est traduite en une fonction lambda récursive. Cependant, en C++ il n'est pas possible [8] de définir une fonction récursive sans préciser son type, ce qui nous oblige à conserver et traduire le type de la fonction. Les règles de traduction des types sont développés en 8.3.2. De plus, nous enveloppons cette fonction lambda récursive dans une fonction lambda pour pouvoir faire une application *inline* et conserver la même règle d'application. La règle de traduction des fonctions récursives CRec est définie ci-dessous :

$$\textsc{CRec}\ \frac{e_{dyn} \rightsquigarrow e_{cpp} \qquad \tau_1 \rightsquigarrow t_{cpp} \qquad \tau_2 \rightsquigarrow {t_{cpp}}'}{\mu_{dyn} f : \tau_1 \rightarrow \tau_2 . \lambda x . e_{dyn} \rightsquigarrow \mathit{RecFunc}(\tau_{cpp}, {\tau_{cpp}}', f, x, e_{cpp})}$$

Avec :

8. Pour notre utilisation des fonctions récursives.

$$
\begin{array}{l}
\mathit{RecFunc}(\tau_{cpp}, \tau_{cpp}', f, x, e_{cpp}) \Leftrightarrow \\
[=](\tau_{cpp}\ x)\{\texttt{auto}\, f = \\
\quad [=](auto\&\ f\_ref, \tau_{cpp} x) -> \tau_{cpp}'\{\texttt{return}\ [f(\%) \mapsto f\_ref(f\_ref, \%)]e_{cpp}; \}; \\
\quad \texttt{return}\ f(f, x); \\
\}
\end{array}
$$

Exemple : Pour mieux comprendre le principe de cette traduction, prenons la fonction qui calcule la valeur de la série de Fibonacci d'un nombre entier. Dans SCOL, on peut écrire cette fonction comme suit :

$$
\begin{array}{l}
\mu_{dyn} fib\colon \texttt{int} \to \texttt{int}.\lambda x.(\\
\quad \texttt{if}_{dyn}\ x = 0\ \texttt{or}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ (fib\ x - 1) + (fib\ x - 2)
\end{array}
$$

Cette fonction deviendra en C++ le code suivant :

```
[=] (int x) {
  auto fib = [=] (auto& fib_ref, int x) -> int {
      return x == 0 || x == 1 ? 1 : fib_ref(fib_ref, x-1) + fib_ref(fib_ref, x-2);
  };
  return fib(fib, x);
}
```

8.5 Exemple

En 5.4, nous avons présenté un scénario de compilation (fig. 8.2). Nous avons déjà effectué le typage statique et l'insertion de type dans le chapitre 7 et l'évaluation statique et la traduction vers SCOL_{dyn} dans le chapitre 6. Nous allons maintenant effectuer la dernière étape de traduction du script et générer le code C++ qui sera exécuté par notre gestionnaire de passes.

Reprenons le code dans SCOL_{dyn} que nous avons obtenu précédemment :

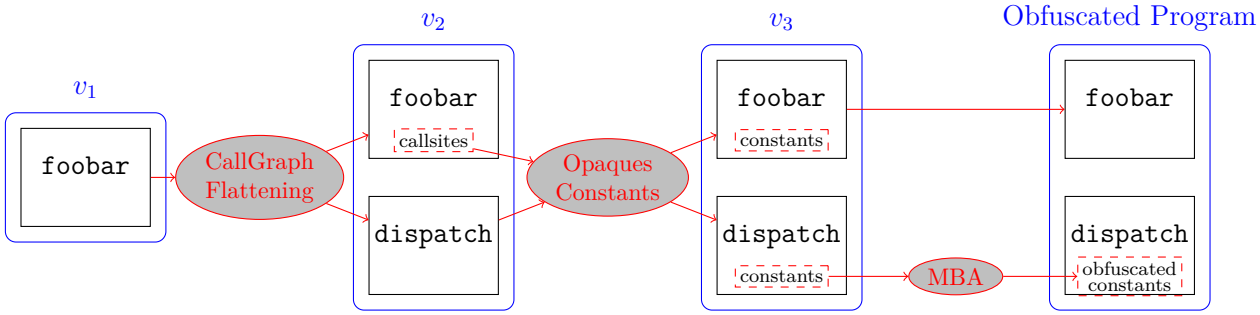

FIGURE 8.2 – Un exemple de chaîne d'obfuscation

$\lambda_{dyn} x.($
  `let` $foobar = (< \alpha$ `with` $\beta = \alpha \oplus$ CGPROT $\wedge\ \alpha <: \langle$ OPT : Pre ; $\partial\top\rangle >\ \pi^3\, x)$ `in`
  $(\pi^1\, x, \pi^2\, x,$
  `RunPass`(OBF.OPQCST,
    $(\pi^1$ `RunPass`(OBF.CGF, $foobar$), $(\pi^3$ `RunPass`(OBF.CGF, $foobar$))),
  `RunPass`(OBF.MBA,
    `RunPass`(OBF.OPQCST,
      $(\pi^2$ `RunPass`(OBF.CGF, $foobar$), $(\pi^3$ `RunPass`(OBF.CGF, $foobar$)))
))

Pour des raisons de lisibilité, nous n'avons pas appliqué le `let`. D'après les règles, que nous avons énoncées dans les chapitres précédents, chaque occurrence de la variable $foobar$ sera remplacée par sa valeur.

L'application des règles de génération de code C++ est directe, mais elle génère du code peu lisible par un humain. Voici, le résultat de l'application des règles dans une version légèrement simplifiée pour faciliter la lecture du code :

```
[=] (const auto& x) {
  return std::make_tuple(std::get<1>(x), std::get<2>(x),

    OpqCst(std::make_tuple(
      std::get<1>(CGF(SCOLCast(/* cast type */, std::get<3>(x)))),
      std::get<3>(CGF(SCOLCast(/* cast type */, std::get<3>(x))))))),
```

```
    MBA(OpqCst(std::make_tuple(
      std::get<1>(CGF(SCOLCast(/* cast type */, std::get<3>(x)))),
      std::get<3>(CGF(SCOLCast(/* cast type */, std::get<3>(x))))))
    )
}
```

Ce code généré est une fonction lambda C++ qui sera intégrée dans notre gestionnaire de passe. Pour exécuter le script, le gestionnaire de passes applique simplement cette fonction sur la liste des fonctions qui composent le programme.

Pour des raisons de lisibilité, nous n'avons pas indiqué ici le type des *casts*. Il s'agit du même type pour chacun des *casts*, qui dans le système de types de SCOL est :

$$\alpha \,\texttt{with}\, \beta = \alpha \oplus \textsc{CGProt} \wedge \alpha <: \langle \textsc{Opt} \,:\, \mathsf{Pre}\,;\, \partial\top \rangle$$

Avec les règles définies en 8.3.4, nous pouvons traduire ce type en C++ :

```
/* cast type = */
SCOLTypeConstraint(SCOLTypeVar("a"), SCOLConstraintAnd(
  SCOLConstraintEq(SCOLTypeVar("b"),
    SCOLTypeRowAdd(SCOLRowVar("a"), SCOLProp("CGProt"))),
  SCOLConstraintSub(SCOLTypeVar("b"),
    SCOLRow([SCOLRowItem("Opt", SCOLRowPre)], SCOLRowTop))
  ))
```

Cet exemple nous permet également de remarquer que notre traduction naïve réalise de nombreuses fois les mêmes opérations. Cela ne change pas la sémantique du script, mais n'est pas très performant. Ce n'est pas un problème pour notre démonstrateur et cela à l'avantage de générer un code ayant une forme proche de la sémantique d'exécution que nous avons décrite dans le chapitre 6.

8.6 Conclusion

Dans ce chapitre, nous avons présenté un démonstrateur pour le gestionnaire de passes et l'exécution, par ce dernier, d'un script d'obfuscation. Cette étape de compilation conclut le processus que nous avons décrit dans le chapitre 5. Notre gestionnaire de passes personnalisé est contenu dans une passe exécutée par le gestionnaire de passes usuel de LLVM. Nous avons développé plusieurs outils pour ce gestionnaire de passes, en particulier, des

classes et des fonctions pour gérer le système de types de SCOL en C++ et l'application des passes.

Ce démonstrateur montre qu'il est possible d'exécuter des scripts d'obfuscation SCOL à l'intérieur du système de compilation LLVM. Il montre que les principes de constructions des chaînes de compilation et d'obfuscation que nous avons décrits dans les chapitres précédents sont applicables en pratique.

Cependant, comme il s'agit d'un démonstrateur, de nombreux choix ont été faits pour la simplicité de l'implémentation au détriment d'autres qualités. Le code C++ est généré naïvement dans un style fonctionnel dans un langage qui n'est pas fonctionnel, ce qui va donner un résultat peu performant. En particulier, la génération de code va créer beaucoup de copie « inutile » et d'exécution redondante de code qui ne seront pas optimisées (manuellement ou par le compilateur C++). De plus, le code est généré sans se préoccuper de sa lisibilité par un humain. Il peut donc être peu lisible (surtout sur de gros scripts), ce qui peut être problématique en particulier pour le débogage. Nous reviendrons sur les limitations de ce démonstrateur et les perspectives d'évolution de celui-ci dans le chapitre 9.

Chapitre 9

Conclusion

Cette thèse propose un système pour la conception et l'exécution de chaînes de compilation prenant en compte les spécificités liées aux attaques *Man-At-The-End*, particulièrement les protections apportées par l'obfuscation de code. Les systèmes de compilation traditionnels sont généralement très efficaces pour l'optimisation des performances des programmes, mais ne prennent pas en compte les problématiques liées à la sécurité. La solution, pour l'application de protections de code dans le processus de compilation, proposée dans cette thèse, est d'utiliser un langage fonctionnel dédié.

Pour cela, nous avons proposé une représentation des passes de compilation, qu'elles soient d'analyse, d'optimisation ou d'obfuscation. Nous avons également proposé une représentation de la représentation des programmes cibles du processus pendant le processus de compilation, qui contient notamment des informations sur les propriétés liées à la sécurité du programme. Ces définitions sont présentées dans le chapitre 5, qui présente également comment ces représentations sont intégrées dans le processus de compilation de SCOL. Nous avons également présenté SCOL, un langage fonctionnel, qui manipule ces représentations pour créer des scripts de compilation et d'obfuscation de programmes (chapitre 6) et son système de types (chapitre 7). Les scripts SCOL sont compilés pour s'intégrer au système de compilation LLVM et au compilateur CLang, pour l'exécution du script sur un programme cible (chapitre 8).

9.1 Contributions de la thèse

Dans cette section, nous allons résumer les contributions et résultats principaux de cette thèse.

Représentation pour les programmes. Nous avons proposé une représentation des programmes dans une chaîne de compilation. Nous avons associé un ensemble de propriétés à chaque programme, et à chaque « morceaux » de programme. Les propriétés sont

des valeurs qui représentent des informations diverses sur le programme. Ces propriétés peuvent être utilisées pour indiquer le niveau de protection du programme, le niveau d'optimisation, les préconditions et postconditions à l'application d'une passe, des résultats d'analyses ou toutes autres informations qui sont utiles pour le processus de compilation. Ces propriétés sont utilisées à la fois dans le processus de compilation afin d'assurer une compilation correcte et spécialisée pour le programme cible, et pour extraire des informations sur le programme final après la compilation pour l'utilisateur, en particulier des informations sur le niveau de protection et la couverture de ces protections. L'ensemble des propriétés n'est pas fixé dans notre représentation, le choix des propriétés est laissé aux utilisateurs et aux développeurs de transformations. Cette représentation est décrite plus précisément dans les chapitres 5 et 8.

Représentation pour les passes. Nous avons également proposé une représentation des passes de compilation. Les passes sont des fonctions contenues dans un système de compilation qui réalisent des transformations (traduction, optimisation, obfuscation...) ou des analyses sur le code. Contrairement aux passes des compilateurs usuels qui sont souvent rigides dans leur structure (leur signature est fixée), les passes de notre processus offrent plus de liberté dans leurs définitions, en particulier les choix de paramètres d'entrée et de sortie. De plus, les passes dans notre approche utilisent une approche fonctionnelle pour l'application de la transformation de code, plutôt que la modification en place des passes traditionnelles.

Les passes agissent également sur les propriétés associées aux programmes qu'elles manipulent : le développeur des passes doit indiquer comment la passe interagit avec les propriétés. Les passes utilisent également les propriétés pour indiquer les conditions d'applications et interagir avec les autres passes dans la chaîne de compilation. Dans les compilateurs usuels, les passes sont souvent des transformations complexes qui peuvent souvent être décomposées en plusieurs transformations simples. Dans notre processus, nous préférons l'utilisation de passes « atomiques » qui représentent ces transformations simples. En effet, l'utilisation de passes simples permet de décrire plus facilement et plus précisément leurs types et les modifications qu'elles effectuent sur les propriétés, ainsi que de créer des transformations composées plus diverses. Cela est rendu possible par la décomposition des passes en passes simples, et leurs compositions pour créer des chaînes complètes.

Cette représentation des passes a pour objectif de pallier les différentes limitations et

problématiques liées à l'utilisation du modèle usuel dans le cadre de la combinaison de transformations d'optimisation et d'obfuscation, que nous avons décrit dans le chapitre 3 et dans [105]. La représentation est décrite en détail dans les chapitres 5 et 8.

SCOL, un langage fonctionnel pour la compilation. Nous avons conçu SCOL (*Safe Control of Obfuscation Language*), un langage fonctionnel dédié à la conception de chaînes de compilation contenant des transformations d'obfuscation. SCOL permet de décrire des chaînes de compilation qui composent des transformations avec plus de finesse et d'expressivité que ce que les systèmes de compilation traditionnels proposent. En effet, SCOL permet un choix précis des lieux d'application et de l'ordre d'application des transformations, ce qui permet d'obtenir un meilleur compromis performance/sécurité qu'une application globale ou aléatoire des passes. SCOL permet de choisir l'ordre et le lieu des applications des passes à la fois manuellement et automatiquement selon le résultat d'autres calculs, comme l'application d'autres passes. Cela permet aux développeurs de programmes et aux développeurs de script SCOL d'apporter des informations « expertes » sur le programme et le script, et au script de déterminer automatiquement la chaîne d'obfuscation à appliquer selon ces informations et le programme d'entrée. SCOL utilise des constructions standards d'un λ-calcul, auxquelles s'ajoutent des constructions pour gérer nos représentations des passes et des programmes. Il adapte également des systèmes d'évaluation partielle et d'évaluation en deux étapes pour s'intégrer dans notre processus d'obfuscation. La syntaxe et la sémantique de SCOL sont décrites en détail dans le chapitre 6.

Un système de types graduel pour SCOL. Le langage SCOL utilise un système de types graduel. L'objectif principal de système de types est d'assurer la correction des scripts d'obfuscation SCOL. En particulier, le système de types utilise des contraintes pour vérifier que les conditions d'applications des passes sont correctes et pour déterminer le type du résultat de l'application de ces passes. Le système de types utilise le typage graduel, car certains termes sont purement dynamiques (les programmes sur lesquels sont appliqués les scripts) tandis que d'autres peuvent être calculés statiquement. Le typage graduel nous permet de vérifier statiquement que les parties statiques du programme sont bien typées et également de vérifier dynamiquement le type des parties dynamique, et ainsi assurer la correction du programme. Cela permet également d'obtenir statiquement des informations (de type) sur les programmes qui sont des entrées valides pour le script et sur

le programme résultat de l'application du script. Pour gérer les propriétés associées aux programmes, le système de types utilise des rangées, que nous avons rendues graduelles et auxquelles nous avons ajouté plusieurs constructions, notamment pour modifier la présence associée à une propriété dans le cadre de contraintes. Le système de types est détaillé dans le chapitre 7.

Intégration à LLVM. Pour montrer les possibilités de notre processus d'obfuscation, nous avons créé un démonstrateur intégré au système de compilation LLVM et au compilateur CLang. Pour cela, le script SCOL est traduit en code C++ qui est compilé à l'intérieur d'une passe qui gère le processus d'exécution de la chaîne de compilation en suivant notre approche. Il gère, notamment, les représentations de passes, de programmes et les propriétés en C++. De plus, vu que notre langage requiert des vérifications dynamiques de types (les *cast*), les types SCOL sont également traduits en C++ et les représentations des constructions en C++ contiennent également un type SCOL, en plus de leur type au sens C++. Enfin, contrairement à SCOL, le langage C++ n'est pas un langage fonctionnel, la traduction doit donc effectuer un changement de paradigme. Cependant, certaines constructions ajoutées au C++ moderne [1] permettent d'écrire du C++ dans un « style » fonctionnel, ce qui rend la traduction plus simple ; la traduction des scripts conserve une forme proche du script original. Ce démonstrateur permet l'exécution de script SCOL et est décrit en détail dans le chapitre 8.

9.2 Limitations et perspectives

Nous allons maintenant étudier des limitations des contributions de cette thèse et des perspectives de travaux futurs. Une des limitations de notre système de compilation est que les programmes cibles sont représentés par un ensemble de fonctions. Cette représentation est suffisante pour beaucoup de transformations, mais une représentation plus fine permettrait une description plus précise de certaines passes. En effet, on pourrait vouloir appliquer des passes sur une entité plus petite qu'une fonction (une boucle, un *basic bloc*, une instruction ...) ou plus grande (un module, le graphe d'appel ...). Adapter notre processus pour gérer plusieurs entités est possible, mais demande de travailler sur certains points. En particulier, comment sont gérées les propriétés dans les différents « niveaux » d'entité : si par exemple une passe modifie les propriétés d'une boucle, comment

1. Particulièrement dans C++14.

les propriétés de la fonction qui contient cette boucle sont-elles affectées ? Comment les propriétés des *basics blocs* contenus dans la boucle sont-elles affectées ? De plus, il est peut être difficile d'identifier les entités de la représentation intermédiaire dans le code source original, notamment les *basics blocs* qui sont des modèles n'existant qu'au niveau de la représentation intermédiaire (contrairement aux fonctions ou aux boucles par exemple).

Une des limitations principales du langage dans son état actuel est que les scripts sont monolithiques, c'est-à-dire que les scripts d'obfuscation sont composés d'un seul « bloc », dans un seul fichier. Par exemple, il n'est pas possible d'utiliser des modules externes ou à un expert d'écrire des bibliothèques et les intégrer dans les scripts SCOL. Cependant, cette modularité est intégrée dans de nombreux langages fonctionnels, et les spécificités de SCOL ne devraient que peu impacter l'intégration dans notre langage. De même, SCOL est un langage monomorphe, et, notamment pour la création de bibliothèques, il pourrait être intéressant de le rendre polymorphique. Notons cependant que le typage graduel permet d'utiliser du polymorphisme sur certaines fonctions.

Pour l'instant, nous n'avons évalué notre système de compilation que sur des scripts SCOL simples, comme les exemples donnés dans cette thèse et dans [105]. Pour montrer l'intérêt de nos contributions, il pourrait être intéressant de les évaluer sur des chaînes de compilation plus complexes et plus grandes. Par exemple, nous pourrions reproduire les chaînes de compilation des obfuscateurs industriels, les chaînes d'optimisation (`-O1`, `-O2` ...) des compilateurs traditionnels ou encore demander à des experts en obfuscation d'écrire des scripts SCOL réalistes.

Un des intérêts de définir formellement le système de types et les différentes sémantiques du langage est de prouver la correction des règles de traduction et que le système de types rend le langage sûr. Nous n'avons pas fait ces preuves, mais un des objectifs de cette contribution étant la sûreté de l'exécution, il sera intéressant de faire ces preuves dans des travaux futurs. Par exemple, en utilisant l'assistant de preuve Coq [29] pour décrire les règles et extraire un compilateur certifié pour SCOL. De plus, le script final généré est en C++ et pour le moment compilé avec CLang, qui n'est pas un compilateur certifié, pour assurer que toute la chaîne est correcte, il sera probablement utile d'utiliser un compilateur certifié comme CompCert [140].

De plus, l'intégration de SCOL dans le système de compilation LLVM est un démonstrateur, il est peut-être fortement amélioré, en particulier d'un point de vue de la performance. En effet, le code C++ est généré naïvement, et créé notamment de nombreuses copies « inutiles », ce qui rend l'exécution peu performante d'un point vu vitesse

d'exécution et utilisation mémoire.

Enfin, notre processus de compilation est intégré au système de compilation LLVM, mais SCOL a été conçu indépendamment de LLVM. Dans le futur, il pourrait être intéressant d'adapter SCOL et son processus de compilation pour l'intégrer à d'autre système de compilation et d'obfuscation, voire de permettre à SCOL d'utiliser plusieurs outils d'obfuscation et de compilation dans un script et de contrôler l'ensemble du processus de compilation. Par exemple, un script SCOL pourrait contrôler un outil effectuant des transformations sources à sources, puis contrôler un compilateur pour compiler le programme et effectuer des transformations sur la représentation intermédiaire et enfin un autre outil pour transformer le binaire. Ce type de script de compilation pourrait également être utile dans d'autres cadres, comme pour de l'optimisation par exemple.

BIBLIOGRAPHIE

[1] Mohsen Ahmadvand, Dennis Fischer, and Sebastian Banescu. « SIP shaker: software integrity protection composition ». In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, pp. 203–214.

[2] A Aho, M Lam, R Sethi, J Ullman, Keith Cooper, Linda Torczon, and S Muchnick. *Compilers: Principles, Techniques and Tools*. 2nd ed. Addison-Wesley, 2007.

[3] Alexander Aiken and Edward L Wimmers. « Type inclusion constraints and type inference ». In: *Proceedings of the conference on Functional programming languages and computer architecture*. 1993, pp. 31–41.

[4] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. « Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions ». In: *Journal of Network and Computer Applications* 48 (2015), pp. 44–57.

[5] Roberto M Amadio and Luca Cardelli. « Subtyping recursive types ». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.4 (1993), pp. 575–631.

[6] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. « A model for self-modifying code ». In: *International Workshop on Information Hiding*. Springer. 2006, pp. 232–248.

[7] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. « Program obfuscation: a quantitative approach ». In: *Proceedings of the 2007 ACM workshop on Quality of protection*. 2007, pp. 15–20.

[8] Andrew Appel. « Deobfuscation is in NP ». In: *Princeton University, Aug* 21 (2002), p. 2.

[9] Kenichi Asai. « Toward introducing binding-time analysis to MetaOCaml ». In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 2016, pp. 97–102.

[10] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. « Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning ». In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3 (2017), pp. 1–28.

[11] *ASObfuscator is an efficient solution designed to obfuscate the source code of C/C++programs*. URL: `http://www.aspack.com/asobfuscator.html`. (accessed: 08.08.2020).

[12] *ASProtect 64 - 64-bit apps protection*. URL: `http://www.aspack.com/asprotect64.html`. (accessed: 08.08.2020).

[13] Mikhail J Atallah and Hoi Chang. *Method and system for tamperproofing software*. US Patent 7,757,097. July 2010.

[14] David Aucsmith. « Tamper resistant software: An implementation ». In: *International Workshop on Information Hiding*. Springer. 1996, pp. 317–333.

[15] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. « Using static analysis to find bugs ». In: *IEEE software* 25.5 (2008), pp. 22–29.

[16] Arini Balakrishnan and Chloe Schulze. « Code obfuscation literature survey ». In: *CS701 Construction of compilers* 19 (2005).

[17] Thomas Ball. « The concept of dynamic analysis ». In: *Software Engineering—ESEC/FSE'99*. Springer. 1999, pp. 216–234.

[18] Thomas Ball and Sriram K Rajamani. « The SLAM project: Debugging system software via static analysis ». In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2002, pp. 1–3.

[19] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. « Code obfuscation against symbolic execution attacks ». In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM. 2016, pp. 189–200.

[20] Sebastian Banescu, Ciprian Lucaci, Benjamin Krämer, and Alexander Pretschner. « VOT4CS: a virtualization obfuscation tool for C ». In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. ACM. 2016, pp. 39–49.

[21] Sebastian Banescu, Martin Ochoa, Nils Kunze, and Alexander Pretschner. « Idea: benchmarking indistinguishability obfuscation–a candidate implementation ». In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2015, pp. 149–156.

[22] Sebastian Banescu, Martin Ochoa, and Alexander Pretschner. « A framework for measuring software obfuscation resilience against automated attacks ». In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE. 2015, pp. 45–51.

[23] Sebastian Banescu and Alexander Pretschner. « A tutorial on software obfuscation ». In: *Advances in Computers*. Vol. 108. Elsevier, 2018, pp. 283–353.

[24] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. « The sorcerer's apprentice guide to fault attacks ». In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.

[25] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. « On the (im) possibility of obfuscating programs ». In: *Annual international cryptology conference*. Springer. 2001, pp. 1–18.

[26] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. « On the (im) possibility of obfuscating programs ». In: *Journal of the ACM (JACM)* 59.2 (2012), pp. 1–48.

[27] Hendrik P Barendregt et al. *The lambda calculus*. Vol. 3. North-Holland Amsterdam, 1984.

[28] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. « Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures ». In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076.

[29] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. « The Coq proof assistant reference manual: Version 6.1 ». In: (1997).

[30] Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, et al. « OCEANS: Optimising compilers for embedded applications ». In: *European Conference on Parallel Processing*. Springer. 1998, pp. 1123–1130.

[31] Thierno Barry, Damien Couroussé, and Bruno Robisson. « Compilation of a countermeasure against instruction-skip fault attacks ». In: *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*. 2016, pp. 1–6.

[32] James Bartusek, Tancrède Lepoint, Fermi Ma, and Mark Zhandry. « New techniques for obfuscating conjunctions ». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, pp. 636–666.

[33] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. « Dynamic analysis of malicious code ». In: *Journal in Computer Virology* 2.1 (2006), pp. 67–77.

[34] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, Nadia Tawbi, et al. « Static detection of malicious code in executable programs ». In: *Int. J. of Req. Eng* 2001.184-189 (2001), p. 79.

[35] Nir Bitansky, Ryo Nishimaki, Alain Passelegue, and Daniel Wichs. « From cryptomania to obfustopia through secret-key functional encryption ». In: *Journal of Cryptology* 33.2 (2020), pp. 357–405.

[36] Nir Bitansky and Vinod Vaikuntanathan. « Indistinguishability obfuscation from functional encryption ». In: *Journal of the ACM (JACM)* 65.6 (2018), pp. 1–37.

[37] Sandrine Blazy. « Specifying and automatically generating a specialization tool for Fortran 90 ». In: *Automated Software Engineering* 7.4 (2000), pp. 345–376.

[38] Manuel Blum and Sampath Kannan. « Designing programs that check their work ». In: *Journal of the ACM (JACM)* 42.1 (1995), pp. 269–291.

[39] Ian Graham Bolton and David Ung. *Partial dead code elimination optimizations for program code conversion*. US Patent 7,543,284. June 2009.

[40] Dan Boneh, Richard A DeMillo, and Richard J Lipton. « On the importance of checking cryptographic protocols for faults ». In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1997, pp. 37–51.

[41] Dan Boneh, Amit Sahai, and Brent Waters. « Functional encryption: Definitions and challenges ». In: *Theory of Cryptography Conference*. Springer. 2011, pp. 253–273.

[42] Viviana Bono and Luigi Liquori. « A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects ». In: *International Workshop on Computer Science Logic*. Springer. 1994, pp. 16–30.

[43] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. « Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on AES ». In: *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*. 2015, pp. 99–103.

[44] Kim B Bruce and Giuseppe Longo. « A modest model of records, inheritance, and bounded quantification ». In: *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design* (1994), pp. 151–196.

[45] Pierrick Brunet, Béatrice Creusillet, Adrien Guinet, and Juan Manuel Martinez. « Epona and the Obfuscation Paradox: Transparent for Users and Developers, a Pain for Reversers ». In: *Proceedings of the 3rd ACM Workshop on Software Protection*. 2019, pp. 41–52.

[46] Finn Brunton and Helen Nissenbaum. *Obfuscation: A user's guide for privacy and protest*. Mit Press, 2015.

[47] Brett Cannon. « Python Developer's Guide Documentation ». In: (2020).

[48] Jan Cappaert and Bart Preneel. « A general model for hiding control flow ». In: *Proceedings of the tenth annual ACM workshop on Digital rights management*. 2010, pp. 35–42.

[49] Matthew Carpenter, Tom Liston, and Ed Skoudis. « Hiding virtualization from attackers and malware ». In: *IEEE Security & Privacy* 5.3 (2007), pp. 62–65.

[50] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. « Rapidly selecting good compiler optimizations using performance counters ». In: *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE. 2007, pp. 185–197.

[51] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. « A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques ». In: *Empirical Software Engineering* 19.4 (2014), pp. 1040–1074.

[52] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. « Towards experimental evaluation of code obfuscation techniques ». In: *Proceedings of the 4th ACM Workshop on Quality of Protection*. 2008, pp. 39–46.

[53] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. « How professional hackers understand protected code while performing attack tasks ». In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE. 2017, pp. 154–164.

[54] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. « Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge ». In: *Empirical Software Engineering* 24.1 (2019), pp. 240–286.

[55] Pinaki Chakraborty. « Fifty years of peephole optimization ». In: *Current Science* (2015), pp. 2186–2190.

[56] David R Chase, Mark Wegman, and F Kenneth Zadeck. « Analysis of pointers and structures ». In: *ACM SIGPLAN Notices* 25.6 (1990), pp. 296–310.

[57] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[58] Brian Chess and Gary McGraw. « Static analysis for security ». In: *IEEE security & privacy* 2.6 (2004), pp. 76–79.

[59] Saurabh Chheda, Kristopher Carver, and Raksit Ashok. *Security of Program Executables and Microprocessors Based on Compiler-Architecture Interaction*. US Patent App. 13/187,645. Apr. 2012.

[60] Frederick Chow, Robert Kennedy, Shin-Ming Liu, Raymond Lo, Peng Tu, and Sun C Chan. *Method, system, and computer program product for performing register promotion via load and store placement optimization within an optimizing compiler*. US Patent 6,128,775. Oct. 2000.

[61] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov. « An approach to the obfuscation of control-flow of sequential computer programs ». In: *International Conference on Information Security*. Springer. 2001, pp. 144–155.

[62] Alonzo Church. « An unsolvable problem of elementary number theory ». In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[63] Alonzo Church. *The calculi of lambda-conversion*. 6. Princeton University Press, 1985.

[64] Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik. « Computer security analysis through decompilation and high-level debugging ». In: *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE. 2001, pp. 375–380.

[65] Christian Collberg. « Tigress: Transformations Index ». In: *University of Arizona* (2015).

[66] Christian S. Collberg and Clark Thomborson. « Watermarking, tamper-proofing, and obfuscation-tools for software protection ». In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 735–746.

[67] Christian S Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. « Distributed application tamper detection via continuous software updates. » In: *ACSAC*. Vol. 12. Citeseer. 2012, pp. 319–328.

[68] Christian S Collberg, Clark Thomborson, and Gregg M Townsend. « Dynamic graph-based software fingerprinting ». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.6 (2007), p. 35.

[69] Christian Sven Collberg, Clark David Thomborson, and Douglas Wai Kok Low. *Obfuscation techniques for enhancing software security*. US Patent 6,668,325. Dec. 2003.

[70] Christian Collberg, Jack Davidson, Roberto Giacobazzi, Yuan Xiang Gu, Amir Herzberg, and Fei-Yue Wang. « Toward digital asset protection ». In: *IEEE Intelligent Systems* 26.6 (2011), pp. 8–13.

[71] Christian Collberg and Clark Thomborson. « Software watermarking: Models and dynamic embeddings ». In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1999, pp. 311–324.

[72] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[73] Christian Collberg, Clark Thomborson, and Douglas Low. « Manufacturing cheap, resilient, and stealthy opaque constructs ». In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 184–196.

[74] Charles Consel. « New insights into partial evaluation: the Schism experiment ». In: *European Symposium on Programming*. Springer. 1988, pp. 236–246.

[75] William R Cook, Walter Hill, and Peter S Canning. « Inheritance is not subtyping ». In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 125–135.

[76] Ingemar J Cox, Matthew L Miller, Jeffrey Adam Bloom, and Chris Honsinger. *Digital watermarking*. Vol. 53. Springer, 2002.

[77] Vijay D'Silva, Mathias Payer, and Dawn Song. « The correctness-security gap in compiler optimization ». In: *2015 IEEE Security and Privacy Workshops*. IEEE. 2015, pp. 73–87.

[78] Min Dai, Christine Eisenbeis, and Sid-Ahmed-Ali Touati. « Load-store optimization for software pipelining ». In: *ACM SIGARCH Computer Architecture News* 28.1 (2000), pp. 3–10.

[79] Jack W Davidson and Jason D Hiser. *System, method and computer program product for protecting software via continuous anti-tampering and obfuscation transforms*. US Patent App. 12/809,627. Feb. 2011.

[80] Rowan Davies. « A temporal-logic approach to binding-time analysis ». In: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1996, pp. 184–195.

[81] Christophe De Vleeschouwer, J-F Delaigle, and Benoit Macq. « Circular interpretation of bijective transformations in lossless watermarking for media asset management ». In: *IEEE Transactions on Multimedia* 5.1 (2003), pp. 97–105.

[82] Alain Deutsch. « Interprocedural may-alias analysis for pointers: Beyond k-limiting ». In: *ACM Sigplan Notices* 29.6 (1994), pp. 230–241.

[83] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. « Type-based alias analysis ». In: *ACM Sigplan Notices* 33.5 (1998), pp. 106–117.

[84] Danny Dolev and Andrew Yao. « On the security of public key protocols ». In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[85] Daniel Dolz and Gerardo Parra. « Using exception handling to build opaque predicates in intermediate code obfuscation techniques ». In: *Journal of Computer Science & Technology* 8 (2008).

[86] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. « The matter of heartbleed ». In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475–488.

[87] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. « Experimental analysis of the laser-induced instruction skip fault model ». In: *Nordic Conference on Secure IT Systems*. Springer. 2019, pp. 221–237.

[88] Java Platform Standard Edition. *Documentation, Oracle, 2015*. 8.

[89] Ben Elliston. *Studying optimisation sequences in the gnu compiler collection*. Tech. rep. Technical Report ZITE8199, School of Information Technology and Electical ..., 2005.

[90] *Epona: code, data and keys protection software | Quarkslab*. URL: `https://quarkslab.com/epona/`. (accessed: 08.08.2020).

[91] Michael D Ernst. « Static and dynamic analysis: Synergy and duality ». In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR. 2003, pp. 24–27.

[92] Ninon Eyrolles. « Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools ». PhD thesis. 2017.

[93] Nicolas Feltman, Carlo Angiuli, Umut A Acar, and Kayvon Fatahalian. « Automatically splitting a two-stage lambda calculus ». In: *European Symposium on Programming*. Springer. 2016, pp. 255–281.

[94] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. « Probabilistic source-level optimisation of embedded programs ». In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. 2005, pp. 78–86.

[95] Jean-loup Gailly and Mark Adler. « Gzip ». In: *rapport no http://www. gzip. org* (2003).

[96] Ronald Garcia, Alison M Clark, and Éric Tanter. « Abstracting gradual typing ». In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 429–442.

[97] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. « Candidate indistinguishability obfuscation and functional encryption for all circuits ». In: *SIAM Journal on Computing* 45.3 (2016), pp. 882–929.

[98] John K Gee, David A Greve, David S Hardin, Allen P Mass, Michael H Masters, Nick M Mykris, and Matthew M Wilding. *Real time processor capable of concurrently running multiple independent JAVA machines*. US Patent 6,374,286. Apr. 2002.

[99] Dan Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, C Pfleeger, J Quarterman, and B Scheier. « CyberInsecurity: The cost of monopoly–how the dominance of Microsoft's products poses a risk to security ». In: *Computer & Communications Industry Association Report* (2003).

[100] Shafi Goldwasser and Guy N Rothblum. « On best-possible obfuscation ». In: *Theory of Cryptography Conference*. Springer. 2007, pp. 194–213.

[101] *Graphing with IDA*. URL: `https://www.hex-rays.com/products/ida/support/tutorials/graphs/`. (accessed: 29.10.2020).

[102] Jonathon Patrick Green, Anjali Doulatram Chandnani, and Simon David Christensen. *Detecting script-based malware using emulation and heuristics*. US Patent 8,997,233. Mar. 2015.

[103] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. « Strong and efficient cache side-channel protection using hardware transactional memory ». In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 217–233.

[104] Serge Guelton. « Building source-to-source compilers for heterogeneous targets ». PhD thesis. 2011.

[105] Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat, and Nicolas Szlifierski. « Combining Obfuscation and Optimizations in the Real World ». In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2018, pp. 24–33.

[106] Adrien Guinet, Ninon Eyrolles, and Marion Videau. « Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions ». In: 2016.

[107] Rajiv Gupta, DA Benson, and Jesse Zhixi Fang. « Path profile guided partial dead code elimination using predication ». In: *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 1997, pp. 102–113.

[108] Bradford E Hampson. *Digital computer system for executing encrypted programs*. US Patent 4,847,902. July 1989.

[109] Ben Hardekopf and Calvin Lin. « The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code ». In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 290–299.

[110] Yong He and M Sc. « Tamperproofing a software watermark by encoding constants ». PhD thesis. University of Auckland, 2002.

[111] Kelly Heffner and Christian Collberg. « The obfuscation executive ». In: *International Conference on Information Security*. Springer. 2004, pp. 428–440.

[112] Laurie J Hendren. *Parallelizing programs with recursive data structures*. Tech. rep. Cornell University, 1990.

[113] William Holder, J Todd McDonald, and Todd R Andel. « Evaluating optimal phase ordering in obfuscation executives ». In: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*. 2017, pp. 1–12.

[114] Thorsten Holz and Frederic Raynal. « Detecting honeypots and other suspicious environments ». In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. IEEE. 2005, pp. 29–36.

[115] Urs Hölzle, Craig Chambers, and David Ungar. « Debugging optimized code with dynamic deoptimization ». In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 1992, pp. 32–43.

[116] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. « Diversification and obfuscation techniques for software security: A systematic literature review ». In: *Information and Software Technology* 104 (2018), pp. 72–93.

[117] François Irigoin, Pierre Jouvelot, and Rémi Triolet. « Semantical interprocedural parallelization: An overview of the PIPS project ». In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. 1991, pp. 143–150.

[118] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[119] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. « Obfuscator-LLVM–software protection for the masses ». In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE. 2015, pp. 3–9.

[120] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. « Emulating emulation-resistant malware ». In: *Proceedings of the 1st ACM workshop on Virtual machine security*. 2009, pp. 11–22.

[121] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. « Exploiting self-modification mechanism for program protection ». In: *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. IEEE. 2003, pp. 170–179.

[122] James C King. « Symbolic execution and program testing ». In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[123] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. « Partial dead code elimination ». In: *ACM SIGPLAN Notices* 29.6 (1994), pp. 147–158.

[124] Ilan Komargodski and Eylon Yogev. « Another step towards realizing random oracles: non-malleable point obfuscation ». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 259–279.

[125] Andreas Krall. « Efficient JavaVM just-in-time compilation ». In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE. 1998, pp. 205–212.

[126] Christopher Kruegel. « Full system emulation: Achieving successful automated dynamic analysis of evasive malware ». In: *Proc. BlackHat USA Security Conference*. 2014, pp. 1–7.

[127] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. « Software countermeasures for control flow integrity of smart card C codes ». In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 200–218.

[128] David Alex Lamb. « Construction of a peephole optimizer ». In: *Software: Practice and Experience* 11.6 (1981), pp. 639–647.

[129] William Landi. « Undecidability of static analysis ». In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[130] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. « SoK: Automated software diversity ». In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 276–291.

[131] James R Larus. « Whole program paths ». In: *ACM SIGPLAN Notices* 34.5 (1999), pp. 259–269.

[132] Tımea László and Ákos Kiss. « Obfuscating C++ programs via control flow flattening ». In: *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30.1 (2009), pp. 3–19.

[133] Chris Lattner. « Introduction to the llvm compiler infrastructure ». In: *Itanium conference and expo*. 2006.

[134] Chris Lattner. « LLVM and Clang: Next generation compiler technology ». In: *The BSD conference*. Vol. 5. 2008.

[135] Chris Arthur Lattner. « LLVM: An infrastructure for multi-stage optimization ». PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[136] Chris Lattner and Vikram Adve. « LLVM: A compilation framework for lifelong program analysis & transformation ». In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[137] Boris Lau and Vanja Svajcer. « Measuring virtual machine detection in malware using DSD tracer ». In: *Journal in Computer Virology* 6.3 (2010), pp. 181–195.

[138] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. « Detecting metamorphic malwares using code graphs ». In: *Proceedings of the 2010 ACM symposium on applied computing*. 2010, pp. 1970–1977.

[139] Seong-Won Lee and Soo-Mook Moon. « Selective just-in-time compilation for client-side mobile javascript engine ». In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. 2011, pp. 5–14.

[140] Xavier Leroy et al. « The CompCert verified compiler ». In: *Documentation and user's manual. INRIA Paris-Rocquencourt* 53 (2012).

[141] Allison Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. « Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption ». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2010, pp. 62–91.

[142] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. « Stochastic optimization of program obfuscation ». In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 221–231.

[143] Kangjie Lu, Siyang Xiong, and Debin Gao. « Ropsteg: program steganography with return oriented programming ». In: *Proceedings of the 4th ACM conference on Data and application security and privacy*. 2014, pp. 265–272.

[144] Anirban Majumdar, Antoine Monsifrot, and Clark Thomborson. « On evaluating obfuscatory strength of alias-based transforms using static analysis ». In: *2006 International Conference on Advanced Computing and Communications*. IEEE. 2006, pp. 605–610.

[145] Ramya Manikyam, J Todd McDonald, William R Mahoney, Todd R Andel, and Samuel H Russ. « Comparing the effectiveness of commercial obfuscators against MATE attacks ». In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. 2016, pp. 1–11.

[146] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. « A taxonomy of self-modifying code for obfuscation ». In: *Computers & Security* 30.8 (2011), pp. 679–691.

[147] William M McKeeman. « Peephole optimization ». In: *Communications of the ACM* 8.7 (1965), pp. 443–444.

[148] Lonny Dean McMichael, Boo Boon Khoo, and Vito Joseph Sabella. *Simultaneous tamper-proofing and anti-piracy protection of software*. US Patent 8,239,967. Aug. 2012.

[149] Nax Paul Mendler. « Inductive types and type constraints in the second-order lambda calculus ». In: *Annals of pure and Applied logic* 51.1-2 (1991), pp. 159–172.

[150] Jason Merrill. « Generic and gimple: A new tree representation for entire functions ». In: *Proceedings of the 2003 GCC Developers' Summit*. Citeseer. 2003, pp. 171–179.

[151] Robin Milner. « A theory of type polymorphism in programming ». In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[152] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly Media, 2013.

[153] John C Mitchell, Furio Honsell, and Kathleen Fisher. « A lambda calculus of objects and method specialization ». In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1993, pp. 26–38.

[154] CVE Mitre. *Heartbleed OpenSSL Bug [CVE-2014-0160]*. 2014.

[155] Torben Ae Mogensen. « Separating binding times in language specifications ». In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 12–25.

[156] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. « The program counter security model: Automatic detection and removal of control-flow side channel attacks ». In: *International Conference on Information Security and Cryptology*. Springer. 2005, pp. 156–168.

[157] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. « Formal verification of a software countermeasure against instruction skip attacks ». In: *Journal of Cryptographic Engineering* 4.3 (2014), pp. 145–156.

[158] Andreas Moser, Christopher Kruegel, and Engin Kirda. « Limits of static analysis for malware detection ». In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE. 2007, pp. 421–430.

[159] Scott A Moskowitz and Marc Cooperman. *Method for stega-cipher protection of computer code*. US Patent 5,745,569. Apr. 1998.

[160] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao, and Lei Shi. « ROPOB: obfuscating binary code via return oriented programming ». In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2017, pp. 721–737.

[161] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[162] Deigo Novillo. « GCC internals ». In: *International Symposium on Code Generation and Optimization (CGO), San Jose, California*. 2007.

[163] Diego Novillo. « GCC an architectural overview, current status, and future directions ». In: *Proceedings of the Linux Symposium*. Vol. 2. 2006, p. 185.

[164] Atsushi Ohori. « A polymorphic record calculus and its compilation ». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.6 (1995), pp. 844–895.

[165] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. « Obfuscation: where are we in anti-DSE protections?(a first attempt) ». In: *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*. 2019, pp. 1–8.

[166] Alfredo Andres Omella. « Methods for virtual machine detection ». In: *Grupo S21sec Gestión SA* (2006).

[167] Mayur Pandey and Suyog Sarda. *LLVM cookbook*. Packt Publishing Ltd, 2015.

[168] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

[169] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[170] Hardware-Based Protections. « A survey of anti-tamper technologies ». In: (2004).

[171] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. « Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines ». In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[172] Geoffrey B Rhoads. *Watermarking methods and media*. US Patent 6,760,463. July 2004.

[173] Valentin Robert and Xavier Leroy. « A formally-verified alias analysis ». In: *International Conference on Certified Programs and Proofs*. Springer. 2012, pp. 11–26.

[174] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. « Return-oriented programming: Systems, languages, and applications ». In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.

[175] Kevin A Roundy and Barton P Miller. « Binary-code obfuscations in prevalent packer tools ». In: *ACM Computing Surveys (CSUR)* 46.1 (2013), pp. 1–32.

[176] Atanas Rountev, Olga Volgin, and Miriam Reddoch. « Static control-flow analysis for reverse engineering of UML sequence diagrams ». In: *ACM SIGSOFT Software Engineering Notes* 31.1 (2005), pp. 96–102.

[177] Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, and Masahide Nakamura. « Empirical study on effects of script minification and HTTP compression for traffic reduction ». In: *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*. IEEE. 2015, pp. 127–132.

[178] Adrian Sampson, Kathryn S McKinley, and Todd Mytkowicz. « Static stages for heterogeneous programming ». In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–27.

[179] Florent Saudel and Jonathan Salwan. « Triton: A dynamic symbolic execution framework ». In: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*. 2015, pp. 31–54.

[180] Jörn-Marc Schmidt and Christoph Herbst. « A practical fault attack on square and multiply ». In: *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2008, pp. 53–58.

[181] Tobias Schneider, Amir Moradi, and Tim Güneysu. « ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks ». In: *Annual International Cryptology Conference*. Springer. 2016, pp. 302–332.

[182] Martin Schoeberl. « A Java processor architecture for embedded real-time systems ». In: *Journal of Systems Architecture* 54.1-2 (2008), pp. 265–286.

[183] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[184] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. « Protecting software through obfuscation: Can it keep pace with progress in code analysis? » In: *ACM Computing Surveys (CSUR)* 49.1 (2016), p. 4.

[185] Juergen Seitz. *Digital watermarking for digital media*. IGI Global, 2005.

[186] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. « Impeding Malware Analysis Using Conditional Code Obfuscation. » In: *NDSS*. 2008.

[187] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. « Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. » In: *NDSS*. 2015.

[188] Jeremy Siek and Matteo Cimini. « The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems ». In: *Symposium on Principles of Programming Languages* (2015), pp. 1–13.

[189] Jeremy Siek, Peter Thiemann, and Philip Wadler. *Blame and coercions: Together again for the first time*. Tech. rep. Technical report, University of Edinburgh, 2014.

[190] Robert Sison and Toby Murray. « Verifying that a compiler preserves concurrent value-dependent information-flow security ». In: *arXiv preprint arXiv:1907.00713* (2019).

[191] Sergei P Skorobogatov and Ross J Anderson. « Optical fault induction attacks ». In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2002, pp. 2–12.

[192] Andrew S Tanenbaum, Hans Van Staveren, and Johan W Stevenson. « Using peephole optimization on intermediate code ». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.1 (1982), pp. 21–36.

[193] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. « Tamper-proofing software watermarks ». In: *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation-Volume 32*. Australian Computer Society, Inc. 2004, pp. 27–36.

[194] Elena Trichina and Roman Korkikyan. « Multi fault laser attacks on protected CRT-RSA ». In: *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2010, pp. 75–86.

[195] Alan M Turing. « Computability and λ-definability ». In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

[196] *VMProtect Software Protection*. URL: `https://vmpsoft.com/`. (accessed: 08.08.2020).

[197] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. *Software tamper resistance: Obstructing static analysis of programs*. Tech. rep. Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[198] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. « Undefined behavior: what happened to my code? » In: *Proceedings of the Asia-Pacific Workshop on Systems*. 2012, pp. 1–7.

[199] *Writing an LLVM Pass*. URL: `https://llvm.org/docs/WritingAnLLVMPass.html`. (accessed: 14.08.2020).

[200] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. « Static control-flow analysis of user-driven callbacks in Android applications ». In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 89–99.

[201] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. « A GPGPU compiler for memory optimization and parallelism management ». In: *ACM Sigplan Notices* 45.6 (2010), pp. 86–97.

[202] Yuval Yarom and Katrina Falkner. « FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack ». In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[203] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. « Information hiding in software with mixed boolean-arithmetic transforms ». In: *International Workshop on Information Security Applications*. Springer. 2007, pp. 61–75.

[204] Lukas Zobernig, Steven D Galbraith, and Giovanni Russello. « When are Opaque Predicates Useful? » In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2019, pp. 168–175.

Annexe A

Syntaxe SCOL

A.1 Syntaxe de SCOL

Cette section présente la syntaxe complète du langage SCOL décrit dans les chapitres 5, 6 et 7. La syntaxe des types (τ) est décrite en D.1.

$$
\begin{array}{lll}
\mathit{main} & ::= & \texttt{script}(x : \tau) = e \\
e & ::= & x \mid \lambda x : \tau_r.e \mid e \,@\, e \\
 & & \mid \texttt{let\_pass}\, x : \tau_r\, \texttt{from}\, \text{PASS}\, \texttt{in}\, e \\
 & & \mid \mathit{constant} \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \\
 & & \mid \mathit{unop}\ e \mid e\ \mathit{binop}\ e \mid e\ \mathit{cop}\ e \mid e\ \mathit{arithOp}\ e \\
 & & \mid e :: e \mid \texttt{hd}\, e \mid \texttt{tl}\, e \mid (e, ..., e) \mid \pi^i\, e \\
 & & \mid \mu f : \tau_r \to \tau_r.\lambda : \tau_r x.e \\
\mathit{constant} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{int} \mid [\,] \\
\mathit{unop} & ::= & \texttt{not} \\
\mathit{binop} & ::= & \texttt{and} \mid \texttt{or} \mid = \\
\mathit{cop} & ::= & < \mid > \mid = \mid \leq \mid \geq \mid \neq \\
\mathit{arithOp} & ::= & + \mid - \mid / \mid * \mid \texttt{mod}
\end{array}
$$

A.2 Syntaxe de SCOL$_{sta}$ et SCOL$_{dyn}$

Cette section présente la syntaxe complète de SCOL$_{sta}$ et SCOL$_{dyn}$ présentée principalement dans le chapitre 6.

$$
\begin{array}{lll}
\mathbb{W} & ::= & sta \mid dyn \mid w \\
main_{sta} & ::= & \texttt{script}(x) = e_{sta} \\
e_{sta} & ::= & x \mid \lambda_{sta} x.e_{sta} \mid e_{sta} \,@_{sta}\, e_{sta} \mid \texttt{let\_pass}\ x\ \texttt{from}\ \textsc{pass}\ \texttt{in}\ e_{sta} \mid <\tau>_{sta}\ \ e_{sta} \\
 & & \mid constant_{sta} \mid \texttt{if}_{sta}\ e_{sta}\ \texttt{then}\ e_{sta}\ \texttt{else}\ e_{sta} \mid unop_{sta}\ e_{sta} \mid e_{sta}\ binop_{sta}\ e_{sta} \\
 & & \mid e_{sta} ::_{sta} e_{sta} \mid \texttt{hd}_{sta}\ e_{sta} \mid \texttt{tl}_{sta}\ e_{sta} \mid (e_{sta}, ..., e_{sta})_{sta} \mid \pi^i_{sta}\ e_{sta} \\
 & & \mid \mu_{sta} f : \tau \to \tau.\lambda x.e_{sta} \\
e_{dyn} & ::= & \texttt{RunPass}(\textsc{pass}, e_{dyn}) \mid e_{dyn} \,@_{dyn}\, e_{dyn} \mid x \\
 & & \mid \texttt{if}_{dyn}\ e_{dyn}\ \texttt{then}\ e_{dyn}\ \texttt{else}\ e_{dyn} \mid unop_{dyn}\ e_{dyn} \mid e_{dyn}\ binop_{dyn}\ e_{dyn} \\
 & & \mid \texttt{hd}_{dyn}\ e_{dyn} \mid \texttt{tl}_{dyn}\ e_{dyn} \mid \pi^i_{dyn}\ e_{dyn} \mid v \mid \textsc{Error} \\
\mathbb{V} :: v & ::= & function \mid program \mid constant_{dyn} \mid <\tau>_{dyn}\ \ e_{dyn} \mid e_{dyn} ::_{dyn} e_{dyn} \mid (e_{dyn}, ..., e_{dyn})_{dyn} \\
function & ::= & \lambda_{dyn} x.e_{dyn} \mid \texttt{pass}(\textsc{pass}) \mid \mu_{dyn} f : \tau \to \tau.\lambda x.e_{dyn} \\
constant_w & ::= & \texttt{true}_w \mid \texttt{false}_w \mid \texttt{int}_w \mid []_w \\
unop_w & ::= & \texttt{not}_w \\
binop_w & ::= & \texttt{and}_w \mid \texttt{or}_w \mid =_w \mid <_w \mid >_w \mid =_w \mid \leq_w \\
 & & \mid \geq_w \mid \neq_w \mid +_w \mid -_w \mid /_w \mid *_w \mid \texttt{mod}_w \\
S_w[\_] & ::= & \_ \,@_w\, e_w \mid e_w \,@_w\, \_ \mid \texttt{if}_w\ \_\ \texttt{then}\ e_w\ \texttt{else}\ e_w \mid \texttt{if}_w\ e_w\ \texttt{then}\ \_\ \texttt{else}\ e_w \\
 & & \mid \texttt{if}_w\ e_w\ \texttt{then}\ e_w\ \texttt{else}\ \_ \mid <\tau>_w\ \ e_w \mid \_ ::_w e_w \mid e_w ::_w \_ \mid \texttt{hd}_w\_ \mid \texttt{tl}_w\_ \\
 & & \mid (e_w, ..., \_, ..., e_w)_w \mid \pi^i_w\ \_ \\
S_{dyn}[\_] & ::= & \texttt{RunPass}(\textsc{pass}, \_)
\end{array}
$$

Annexe B

Sémantique d'évaluation de SCOL

B.1 Sémantique d'évaluation de SCOL

Cette section présente la sémantique d'évaluation de SCOL. Ces règles sont de la forme : $e \Rightarrow v$.

$$\textsc{Script}\ \frac{[x \mapsto v]e \Rightarrow_{sta} v'}{(\texttt{script}(x) = e)\, v \Longrightarrow v'}$$

$$\textsc{ScriptError}\ \frac{[x \mapsto v]e \Rightarrow_{sta} \textsc{Error}}{(\texttt{script}(x) = e)\, v \Longrightarrow \textsc{Error}}$$

$$\textsc{App}\ \frac{e_w \Rightarrow_w \lambda_{dyn} x.e_{dyn}' \qquad e_w'' \Rightarrow_w v \qquad [x \mapsto v]e_{dyn}' \Rightarrow_{dyn} v'}{e_w \mathbin{@}_w e_w'' \Rightarrow_w v'}$$

$$\textsc{AppError}\ \frac{e_w \Rightarrow_w \lambda_{dyn} x.e_{dyn}' \qquad e_w'' \Rightarrow_w v \qquad [x \mapsto v]e_{dyn}' \Rightarrow_{dyn} \textsc{Error}}{e_w \mathbin{@}_w e_w'' \Rightarrow_w \textsc{Error}}$$

$$\textsc{LetPass}\ \frac{[x \mapsto \texttt{pass}(\textsc{pass})]e \Rightarrow_{sta} v}{\texttt{let\_pass}\ x\ \texttt{from}\ \textsc{pass}\ \texttt{in}\ e \Rightarrow_{sta} v}$$

$$\textsc{RunPass}\ \frac{e_{dyn} \Rightarrow_{dyn} v \qquad \{\textsc{pass}, p, cond\} \in \mathbb{P} \qquad cond(v)}{\texttt{RunPass}(\textsc{pass}, e_{dyn}) \Rightarrow_{dyn} p(v)}$$

$$\textsc{AppPass}\ \frac{e_w \Rightarrow_w \texttt{pass}(\textsc{pass}) \qquad e_w'' \Rightarrow_w v \qquad \{\textsc{pass}, p, cond\} \in \mathbb{P} \qquad cond(v)}{e_w \mathbin{@}_w e_w'' \Rightarrow_w p(v)}$$

$$\textsc{PassError}\ \frac{\textsc{pass} \notin dom(\mathbb{P})}{\texttt{RunPass}(\textsc{pass}, e_{dyn}) \Rightarrow_{dyn} \textsc{Error}}$$

$$\textsc{PassErrorII}\ \frac{e_{dyn} \Rightarrow_{dyn} v \qquad \{\textsc{pass}, p, cond\} \in \mathbb{P} \qquad \neg cond(v)}{\texttt{RunPass}(\textsc{pass}, e_{dyn}) \Rightarrow_{dyn} \textsc{Error}}$$

$$\textsc{Cast}\ \frac{e_w \Rightarrow_w v \qquad \texttt{typeof}(e_w) = \tau \qquad \tau <: \tau'}{(<\tau'> \ e_w) \Rightarrow_w v}$$

$$\textsc{CastError}\ \frac{\texttt{typeof}(e_w) = \tau \qquad \neg(\tau <: \tau')}{(<\tau'> \ e_w) \Rightarrow_w \textsc{Error}}$$

$$\textsc{True}\ \texttt{true}_w \Rightarrow_w \texttt{true}$$

$$\textsc{False}\ \texttt{false}_w \Rightarrow_w \texttt{false}$$

$$\textsc{NotTrue}\ \frac{e_w \Rightarrow_w \texttt{true}}{\texttt{not}e_w \Rightarrow_w \texttt{false}}$$

$$\textsc{NotFalse}\ \frac{e_w \Rightarrow_w \texttt{false}}{\texttt{not}e_w \Rightarrow_w \texttt{true}}$$

$$\textsc{NotError}\ \frac{e_w \Rightarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{\texttt{not}e_w \Rightarrow_w \textsc{Error}}$$

$$\textsc{Binop}\ \frac{e_w \Rightarrow_w v \qquad {e_w}' \Rightarrow_w v' \qquad op \in binop \qquad \texttt{Compute}(v\,op\,v') = v''}{e_w\,op\,e_w \Rightarrow_w v''}$$

$$\textsc{IfTrue}\ \frac{e_w \Rightarrow_w \texttt{true} \qquad {e_w}' \Rightarrow_w v'}{\texttt{if}_w\, e_w\, \texttt{then}\, {e_w}'\, \texttt{else}\, {e_w}'' \Rightarrow_w v'}$$

$$\textsc{IfFalse}\ \frac{e_w \Rightarrow_w \texttt{false} \qquad {e_w}'' \Rightarrow_w v''}{\texttt{if}_w\, e_w\, \texttt{then}\, {e_w}'\, \texttt{else}\, {e_w}'' \Rightarrow_w v''}$$

$$\textsc{IfError}\ \frac{e_w \Rightarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{\texttt{if}_w\, e_w\, \texttt{then}\, {e_w}'\, \texttt{else}\, {e_w}'' \Rightarrow_w \textsc{Error}}$$

$$\textsc{List}\ \frac{e_w \Rightarrow_w v \qquad e_w{}' \Rightarrow_w v'}{e_w :: e_w{}' \Rightarrow_w v :: v'} \qquad \textsc{Hd}\ \frac{e_w \Rightarrow_w v :: v'}{\texttt{hd}e_w \Rightarrow_w v}$$

$$\textsc{Tl}\ \frac{e_w \Rightarrow_w v :: v'}{\texttt{tl}e_w \Rightarrow_w v'} \qquad \textsc{HdError}\ \frac{e_w \Rightarrow_w v \qquad v \neq v' :: v''}{\texttt{hd}e_w \Rightarrow_w \textsc{Error}}$$

$$\textsc{TlError}\ \frac{e_w \Rightarrow_w v \qquad v \neq v' :: v''}{\texttt{tl}e_w \Rightarrow_w \textsc{Error}} \qquad \textsc{Tuple}\ \frac{\forall i \in [1,n], e_i \Rightarrow_w v_i}{(e_1,...,e_n) \Rightarrow_w (v_1,...,v_n)}$$

$$\textsc{TupleProj}\ \frac{e_w \Rightarrow_w (v_1,...,v_n) \qquad 0 < i < n+1 \vee}{\pi^i_w e_w \Rightarrow_w v_i}$$

$$\textsc{TupleError}\ \frac{e_w \Rightarrow_w (v_1,...,v_n) \qquad i > n \vee i < 1}{\pi^i_w e_w \Rightarrow_w \textsc{Error}}$$

$$\textsc{TupleErrorII}\ \frac{e_w \Rightarrow_w v \qquad v \neq (v_1,...,v_n)}{\pi^i_w e_w \Rightarrow_w \textsc{Error}}$$

$$\textsc{Rec}\ \frac{e_w \Rightarrow_w \mu f \lambda x.e_{dyn} \qquad e_w{}' \Rightarrow_w v \qquad [f \mapsto \mu f \lambda x.e_{dyn}, x \mapsto v]e_{dyn} \Rightarrow_{dyn} v'}{e_w \,@_w\, e_w{}' \Rightarrow_w v'}$$

Annexe C

Sémantique de traduction de SCOL

C.1 Traduction de SCOL$_{sta}$ vers SCOL$_{dyn}$

Cette section présente la sémantique de la traduction de SCOL$_{sta}$ vers SCOL$_{dyn}$. Les règles sont de la forme : $e_w \Rightarrow_w e_{dyn}$, où w est une variable désignant sta où dyn.

$$\textsc{script}\ \frac{\{x \mapsto x\} \vdash_{sta} e \Downarrow_{sta} e_{dyn}}{\texttt{script}(x) = e \Downarrow \lambda_{dyn} x.e_{dyn}}$$

$$\textsc{scriptError}\ \frac{\{x \mapsto x\} \vdash_{sta} e \Downarrow_{sta} \textsc{Error}}{\texttt{script}(x) = e \Downarrow \textsc{Error}} \qquad \textsc{Var}\ \frac{\{x \mapsto e_{dyn}\} \in E}{E \vdash_w x \Downarrow_w e_{dyn}}$$

$$\textsc{VarError}\ \frac{x \notin dom(E)}{E \vdash_w x \Downarrow_w \textsc{Error}} \qquad \textsc{Lambda}\ \frac{E \cup \{x \mapsto x\} \vdash_w e_w \Downarrow_w e_{dyn}}{E \vdash_w \lambda_w x.e_w \Downarrow_w \lambda_{dyn} x.e_{dyn}}$$

$$\textsc{LambdaError}\ \frac{E \cup \{x \mapsto x\} \vdash_w e_w \Downarrow_w \textsc{Error}}{E \vdash_w \lambda_w x.e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{StaticApp}\ \frac{E \vdash_w e_1 \Downarrow_w \lambda_{dyn} x.e_{dyn} \qquad E \vdash_w e_2 \Downarrow_w v \qquad E \cup \{x \to v\} \vdash_{dyn} e_{dyn} \Downarrow_{dyn} e'_{dyn}}{E \vdash_w e_1 \mathbin{@}_w e_2 \Downarrow_w e'_{dyn}}$$

$$\textsc{Context}\ \frac{E \vdash_w e \Downarrow_w e' \qquad e' \notin \mathbb{V}}{E \vdash_w S_w[e] \Downarrow_w S_{dyn}[e']} \qquad \textsc{ContextError}\ \frac{E \vdash_w e \Downarrow_w \textsc{Error}}{E \vdash_w S_w[e] \Downarrow_w \textsc{Error}}$$

$$\textsc{AppErrorI}\ \frac{e_w \Rightarrow_w \lambda_{dyn} x.e_{dyn}{}' \qquad e_w{}'' \Rightarrow_w v \qquad [x \mapsto v]e_{dyn}{}' \Rightarrow_{dyn} \textsc{Error}}{e_w \mathbin{@}_w e_w{}'' \Rightarrow_w \textsc{Error}}$$

$$\textsc{AppErrorII}\ \frac{e_w'' \Rightarrow_w \textsc{Error}}{e_w \mathbin{@}_w e_w'' \Rightarrow_w \textsc{Error}} \qquad \textsc{AppErrorIII}\ \frac{e_w \Rightarrow_w \textsc{Error}}{e_w \mathbin{@}_w e_w'' \Rightarrow_w \textsc{Error}}$$

$$\textsc{PassApp}\ \frac{E \vdash_w e_1 \Downarrow_w \texttt{pass}(\textsc{pass}) \qquad E \vdash_w e_2 \Downarrow_w e_{dyn}}{E \vdash_w e_1 \mathbin{@}_w e_2 \Downarrow_w \texttt{RunPass}(\textsc{pass}, e_{dyn})}$$

$$\textsc{LetPass}\ \frac{E \cup \{x \mapsto \texttt{pass}(\textsc{pass})\} \vdash_{sta} e \Downarrow_{sta} e_{dyn}}{E \vdash_{sta} \texttt{let\_pass}\ x\ \texttt{from}\ \textsc{pass}\ \texttt{in}\ e \Downarrow_{sta} e_{dyn}}$$

$$\textsc{NotTrue}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{true}_{dyn}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \texttt{false}_{dyn}} \qquad \textsc{NotFalse}\ \frac{E \vdash e_w \Downarrow_w \texttt{false}_{dyn}}{E \vdash \texttt{not}_w\, e_w \Downarrow_w \texttt{true}_{dyn}}$$

$$\textsc{DynamicNot}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} \qquad e_{dyn} \notin \mathbb{V}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \texttt{not}_{dyn}\, e_{dyn}}$$

$$\textsc{NotError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{E \vdash_w \texttt{not}_w\, e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{ReducBinOp}\ \frac{\begin{array}{c} E \vdash_w e_w \Downarrow e_{dyn} \\ E \vdash e_w' \Downarrow e_{dyn}' \qquad op \in binop_w \qquad \texttt{Red}(e_{dyn}\, op\, e_{dyn}') = e_{dyn}'' \end{array}}{E \vdash_w e_w\, op_w\, e_w' \Downarrow_w e_{dyn}''}$$

$$\textsc{IfTrue}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{true}_{dyn} \qquad E \vdash e_w' \Downarrow e_{dyn}}{E \vdash_w \texttt{if}\ e_w\ \texttt{then}\ e_w'\ \texttt{else}\ e_w'' \Downarrow_w e_{dyn}}$$

$$\textsc{IfFalse}\ \frac{E \vdash_w e_w \Downarrow_w \texttt{false}_{dyn} \qquad E \vdash_w e_w'' \Downarrow_w e_{dyn}}{E \vdash_w \texttt{if}\ e_w\ \texttt{then}\ e_w'\ \texttt{else}\ e_w'' \Downarrow_w e_{dyn}}$$

$$\textsc{IfError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \notin \{\texttt{true}_{dyn}, \texttt{false}_{dyn}\}}{E \vdash_w \texttt{if}_w\ e_w\ \texttt{then}\ e_w'\ \texttt{else}\ e_w'' \Downarrow \textsc{Error}}$$

$$\textsc{StaticHd}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} ::_{dyn} e_{dyn}' \qquad e_{dyn} \neq \textsc{Error} \qquad e_{dyn}' \neq \textsc{Error}}{E \vdash_w \mathtt{hd}_w\, e_w \Downarrow_w e_{dyn}}$$

$$\textsc{StaticTl}\ \frac{E \vdash_w e_w \Downarrow_w e_{dyn} ::_{dyn} e_{dyn}' \qquad e_{dyn} \neq \textsc{Error} \qquad e_{dyn}' \neq \textsc{Error}}{E \vdash_w \mathtt{tl}_w\, e_w \Downarrow_w e_{dyn}'}$$

$$\textsc{HdError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq e_{dyn} ::_{dyn} e_{dyn}'}{E \vdash_w \mathtt{hd}_w\, e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{TlError}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq e_{dyn} ::_{dyn} e_{dyn}'}{E \vdash_w \mathtt{tl}_w\, e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{StaticTupleProj}\ \frac{E \vdash_w e_w \Downarrow_w (e^1_{dyn}, \ldots, e^i_{dyn}, \ldots, e^n_{dyn})_{dyn}}{E \vdash_w \pi^i_w\, e_w \Downarrow_w e^i_{dyn}}$$

$$\textsc{TupleError}\ \frac{E \vdash_w e_w \Downarrow_w (e^1_{dyn}, \ldots, e^n_{dyn})_{dyn} \qquad i > n \vee i < 1}{E \vdash_w \pi^i_w e \Downarrow_w \textsc{Error}}$$

$$\textsc{TupleErrorII}\ \frac{E \vdash_w e_w \Downarrow_w v \qquad v \neq (e^1_{dyn}, \ldots, e^n_{dyn})_{dyn}}{E \vdash_w \pi^i_w e_w \Downarrow_w \textsc{Error}}$$

$$\textsc{StaticRec}\ \frac{\begin{array}{c} E \vdash_{sta} e_{sta} \Downarrow_{sta} \mu_{dyn} f.\lambda x.e_{dyn} \\ E \vdash_{sta} e_{sta}' \Downarrow_{sta} v \qquad E \cup \{f \mapsto \mu_{dyn} f.\lambda x.e_{dyn}, x \mapsto v\} \vdash_{dyn} e_{dyn} \Downarrow e_{dyn}' \end{array}}{E \vdash_{sta} e_{sta}\ @_{sta}\ e_{sta}' \Downarrow_{sta} e_{dyn}'}$$

$$\textsc{DynamicRec}\ \frac{E \vdash_{dyn} e_{dyn} \Downarrow_{dyn} \mu_{dyn} f.\lambda x.e_{dyn}' \qquad E \vdash_{dyn} e_{dyn}'' \Downarrow_{dyn} e_{dyn}'''}{E \vdash_{dyn} e_{dyn}\ @_{dyn}\ e_{dyn}'' \Downarrow_{dyn} \mu_{dyn} f.\lambda x.e_{dyn}'\ @_{dyn}\ e_{dyn}'''}$$

$$\textsc{Mu}\ \frac{E \cup \{f \mapsto f, x \mapsto x\} \vdash_w e_w \Downarrow_w e_{dyn}}{E \vdash_w \mu_w f.\lambda x.e_w \Downarrow_w \mu_{dyn} f.\lambda x.e_{dyn}}$$

Annexe D

Insertion de cast SCOL

D.1 Systeme de types

Cette section présente la syntaxe du système de types de SCOL, qui est détaillé dans le chapitre 7.

$$
\begin{array}{lll}
\tau & ::= & \tau_r \mid ? \\
\tau_r & ::= & \tau_r \to \tau_r \mid \alpha \mid \tau_r \,\texttt{with}\, C_{main} \mid < row > \\
 & & \mid \texttt{bool} \mid \texttt{int} \mid \tau\,\texttt{list} \mid \tau * ... * \tau \mid \texttt{TupleSize}(i) \\
row & ::= & \textsc{Prop} : pres; row \mid \partial pres \\
pres & ::= & \mathsf{Pre} \mid \mathsf{Abs} \mid \top \\
C_{main} & ::= & C_{main} \bigvee C_{main} \mid C \\
C & ::= & C \wedge C \mid \tau_r <: \tau_r \mid \tau_r = \tau_r \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\, C \mid (C)
\end{array}
$$

D.2 Insertion des *casts*

Cette section présente les règles de typage et d'insertion de *casts*. Ces règles sont de la forme : $e \Rightarrow e_{cast} : \tau$, avec e un terme de SCOL, e_{cast} la traduction dans SCOL_{cast} et τ le type du terme.

$$
\textsc{ScriptObf}\ \frac{x : \tau \vdash e \Rightarrow e_{cast} : \tau'}{\texttt{script}(x : \tau) = e \Rightarrow \texttt{script}(x) = e_{cast} : \tau'}
$$

$$
\textsc{App1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'' \qquad \tau'' <: \tau}{\Gamma \vdash e_1 \,@\, e_2 \Rightarrow {e_{cast}}^1 \,@\, {e_{cast}}^2 : \tau'}
$$

$$
\textsc{App2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'' \qquad \tau'' \sim \tau \qquad \neg(\tau'' <: \tau)}{\Gamma \vdash e_1 \,@\, e_2 \Rightarrow {e_{cast}}^1 \,@\, ((< \tau > \ {e_{cast}}^2)) : \tau'}
$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow x : \tau} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow e_{cast} : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \lambda x . e_{cast} : \tau_1 \to \tau_2}$$

$$\textsc{LetPass}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow e_{cast} : \tau_2}{\Gamma \vdash \texttt{let\_pass}\, x : \tau_1\, \texttt{from PASS in}\, e \Rightarrow \texttt{let\_pass}\, x\, \texttt{from PASS in}\, e_{cast} : \tau_2}$$

$$\textsc{App3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau_1 \to \tau_2\, \texttt{with}\, C \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau \quad \tau <: \tau' \quad \tau' \in S[C](\alpha)}{\Gamma \vdash e_1 \mathbin{@} e_2 \Rightarrow {e_{cast}}^1 \mathbin{@} {e_{cast}}^2 : \tau_2\, \texttt{with}\, [\tau_1 \mapsto \tau]C}$$

$$\textsc{App4}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau_1 \to \tau_2\, \texttt{with}\, C \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau \quad \nexists \tau_3, \tau <: \tau_3, \tau_3 \in S[C](\tau_1) \quad \exists \tau' \in S[C](\tau_1), \tau \sim \tau'}{\Gamma \vdash e_1 \mathbin{@} e_2 \Rightarrow {e_{cast}}^1 \mathbin{@} (< \tau_1\, \texttt{with}\, C > \ {e_{cast}}^2) : \tau_2\, \texttt{with}\, [\tau_1 \mapsto \tau]C}$$

$$\textsc{True}\ \Gamma \vdash \texttt{true} \Rightarrow \texttt{true} : \texttt{bool} \qquad \textsc{False}\ \Gamma \vdash \texttt{false} \Rightarrow \texttt{false} : \texttt{bool}$$

$$\textsc{Not1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e \Rightarrow \texttt{not}\ e_{cast} : \texttt{bool}}$$

$$\textsc{Not2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{not}\ e \Rightarrow \texttt{not}(< \texttt{bool} > \ e_{cast}) : \texttt{bool}}$$

$$\textsc{Binop1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{bool}}{\Gamma \vdash e_1\, binop\, e_2 \Rightarrow {e_{cast}}^1\, binop\, {e_{cast}}^2 : \texttt{bool}}$$

$$\textsc{Binop2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \quad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{bool}}{\Gamma \vdash e_1\, binop\, e_2 \Rightarrow (< \texttt{bool} > \ {e_{cast}}^1)\, binop\, {e_{cast}}^2 : \texttt{bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : \tau_2 \qquad \exists \tau, \tau_1 <: \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ {e_{cast}}^1\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ e_{cast3} : \tau}$$

$$\textsc{If1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : \tau_2 \qquad \exists \tau, \tau_1 <: \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ (< \texttt{bool} >\ {e_{cast}}^1)\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ e_{cast3} : \tau}$$

$$\textsc{If2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : \tau_2 \qquad \exists \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ {e_{cast}}^1\ \texttt{then}\ (< \tau >\ {e_{cast}}^2)\ \texttt{else}\ e_{cast3} : \tau}$$

$$\textsc{If3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : ? \qquad \exists \tau, \tau_1 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ {e_{cast}}^1\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ (< \tau >\ e_{cast3}) : \tau}$$

$$\textsc{If23}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : ?}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ {e_{cast}}^1\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ e_{cast3} : ?}$$

$$\textsc{If123}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : ?}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ (< \texttt{bool} >\ {e_{cast}}^1)\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ e_{cast3} : ?}$$

$$\textsc{If12}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ? \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : \tau_2 \qquad \exists \tau, \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ (< \texttt{bool} >\ {e_{cast}}^1)\ \texttt{then}\ (< \tau >\ {e_{cast}}^2)\ \texttt{else}\ e_{cast3} : \tau}$$

$$\textsc{If13}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau_1 \qquad \Gamma \vdash e_3 \Rightarrow e_{cast3} : ? \qquad \exists \tau, \tau_1 <: \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ (< \texttt{bool} >\ {e_{cast}}^1)\ \texttt{then}\ {e_{cast}}^2\ \texttt{else}\ (< \tau >\ e_{cast3}) : \tau}$$

$$\textsc{Int}\ \Gamma \vdash int \Rightarrow int : \texttt{int} \qquad \textsc{Cop1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, cop\, e_2 \Rightarrow {e_{cast}}^1\, cop\, {e_{cast}}^1 : \texttt{bool}}$$

$$\textsc{Cop2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, cop\, e_2 \Rightarrow (<\texttt{int}>\ {e_{cast}}^1)\, cop\, {e_{cast}}^1 : \texttt{bool}}$$

$$\textsc{ArithOp1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, arithOp\, e_2 \Rightarrow {e_{cast}}^1\, arithOp\, {e_{cast}}^1 : \texttt{int}}$$

$$\textsc{ArithOp2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \texttt{int}}{\Gamma \vdash e_1\, arithOp\, e_2 \Rightarrow (<\texttt{int}>\ {e_{cast}}^1)\, arithOp\, {e_{cast}}^1 : \texttt{int}}$$

$$\textsc{ListEmpty}\ \Gamma \vdash [] \Rightarrow [] : ?\,\texttt{list}$$

$$\textsc{ListCons1}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : ?\,\texttt{list} \qquad \tau \neq ?}{\Gamma \vdash e_1 :: e_2 \Rightarrow {e_{cast}}^1 ::(<\tau\texttt{list}>\ {e_{cast}}^2) : \tau'\,\texttt{list}}$$

$$\textsc{ListCons2}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : \tau \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'\,\texttt{list} \qquad \exists \tau'', \tau <: \tau'', \tau' <: \tau''}{\Gamma \vdash e_1 :: e_2 \Rightarrow {e_{cast}}^1 :: {e_{cast}}^2 : \tau''\,\texttt{list}}$$

$$\textsc{ListCons3}\ \frac{\Gamma \vdash e_1 \Rightarrow {e_{cast}}^1 : ? \qquad \Gamma \vdash e_2 \Rightarrow {e_{cast}}^2 : \tau'\,\texttt{list} \qquad \tau' \neq ?}{\Gamma \vdash e_1 :: e_2 \Rightarrow (<\tau'>\ {e_{cast}}^1) :: {e_{cast}}^2 : \tau'\,\texttt{list}}$$

$$\textsc{Tl1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau\ \texttt{list}}{\Gamma \vdash \texttt{tl}\, e \Rightarrow \texttt{tl}\, e_{cast} : \tau\ \texttt{list}} \qquad \textsc{Hd1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau\ \texttt{list}}{\Gamma \vdash \texttt{hd}\, e \Rightarrow \texttt{hd}\, e_{cast} : \tau}$$

$$\textsc{Tl2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{tl}\, e \Rightarrow \texttt{tl}\, (<?\,\texttt{list} >\ e_{cast}) : ?\,\texttt{list}}$$

$$\textsc{Hd2}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \texttt{hd}\, e \Rightarrow \texttt{hd}\, (<?\,\texttt{list} >\ e_{cast}) : ?}$$

$$\textsc{TupleCons}\ \frac{\forall i \in [1, n], \Gamma \vdash e_i \Rightarrow e^i_{cast} : \tau_i}{\Gamma \vdash (e_1, ..., e_n) \Rightarrow {e_{cast}}^1, ..., e^n_{cast} : \tau_1 * ... * \tau_n}$$

$$\textsc{Proj}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : \tau_1 * ... * \tau_n}{\Gamma \vdash \pi_i\, e \Rightarrow \pi_i\, e_{cast} : \tau_i}$$

$$\textsc{Proj1}\ \frac{\Gamma \vdash e \Rightarrow e_{cast} : ?}{\Gamma \vdash \pi_i\, e \Rightarrow \pi_i\, (<\texttt{TupleSize}(i) >\ e_{cast}) : ?}$$

$$\textsc{Rec}\ \frac{\Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e \Rightarrow e_{cast} : \tau_3}{\Gamma \vdash \mu f : \tau_1 \to \tau_2 \lambda x : \tau_1 . e \Rightarrow \mu f : \tau_1 \to \tau_2 \lambda x . e_{cast} : \tau_1 \to \tau_3}$$

Annexe E

Traduction de SCOL vers C++

E.1 Règles de traduction

Cette section présente les règles de traduction de SCOL_{dyn} vers C++. Les règles sont de la forme : $e_{dyn} \rightsquigarrow e_{cpp}$ où e_{dyn} est un terme de SCOL_{dyn} et e_{cpp} un terme en C++.

$$\textsc{CRunPass}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{RunPass}(\text{PASS}, e_{dyn}) \rightsquigarrow \text{PASS}(e_{cpp})} \qquad \textsc{CPass}\ \texttt{pass}(\text{PASS}) \rightsquigarrow \text{PASS}$$

$$\textsc{CVar}\ x \rightsquigarrow x \qquad \textsc{CApp}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn} @_{dyn} e^2_{dyn} \rightsquigarrow e^1_{cpp}(e^2_{cpp})}$$

$$\textsc{CAbs}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\lambda_{dyn} x.e_{dyn} \rightsquigarrow [\&](\texttt{const auto\& x})\{\texttt{return}\ e_{cpp};\}}$$

$$\textsc{CCast}\ \frac{\tau \hookrightarrow \texttt{SCOLType} \qquad e_{dyn} \rightsquigarrow e_{cpp}}{(<\tau>\ \ e_{dyn}) \rightsquigarrow \texttt{SCOLCast}(\texttt{SCOLType}, e_{cpp})}$$

$$\textsc{CTrue}\ \texttt{true} \rightsquigarrow \texttt{true} \qquad \textsc{CFalse}\ \texttt{false} \rightsquigarrow \texttt{false} \qquad \textsc{CInt}\ \texttt{int} \rightsquigarrow \texttt{int}$$

$$\textsc{CUnop}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{unop_{dyn}\ e_{dyn} \rightsquigarrow unop_{cpp}\ e_{cpp}}$$

$$\textsc{CBinop}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn}\ binop_{dyn}\ e^2_{dyn} \rightsquigarrow e^1_{cpp}\ binop_{cpp}\ e^2_{cpp}}$$

$$\textsc{CIf}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp} \qquad e^3_{dyn} \rightsquigarrow e^3_{cpp}}{\texttt{if}_{dyn}\ e^1_{dyn}\ \texttt{then}\ e^2_{dyn}\ \texttt{else}\ e^3_{dyn} \rightsquigarrow e^1_{cpp}\ ?\ e^2_{cpp}\ :\ e^3_{cpp}}$$

$$\textsc{CEmptyList}\ [\,]_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{EmptyList}()$$

$$\textsc{CHd}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{hd}\, e_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{hd}(e_{cpp})} \qquad \textsc{CTl}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\texttt{tl}\, e_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{tl}(e_{cpp})}$$

$$\textsc{CCons}\ \frac{e^1_{dyn} \rightsquigarrow e^1_{cpp} \qquad e^2_{dyn} \rightsquigarrow e^2_{cpp}}{e^1_{dyn} ::_{dyn} e^2_{dyn} \rightsquigarrow \texttt{SCOLList} :: \texttt{cons}(e^1_{cpp}, e^2_{cpp})}$$

$$\textsc{TupleProj}\ \frac{e_{dyn} \rightsquigarrow e_{cpp}}{\pi^i_{dyn}\, e_{dyn} \rightsquigarrow \texttt{std} :: \texttt{get} < \texttt{i} > (e_{cpp})}$$

$$\textsc{TupleCons}\ \frac{\forall i \in [1, n], e^i_{dyn} \rightsquigarrow e^i_{cpp}}{(e^1_{dyn}, ..., e^n_{dyn})_{dyn} \rightsquigarrow \texttt{std} :: \texttt{make\_tuple}(e^1_{cpp}, ..., e^2_{cpp})}$$

$$\textsc{CRec}\ \frac{e_{dyn} \rightsquigarrow e_{cpp} \qquad \tau_1 \rightsquigarrow t_{cpp} \qquad \tau_2 \rightsquigarrow t_{cpp}{}'}{\mu_{dyn} f : \tau_1 \to \tau_2 . \lambda x . e_{dyn} \rightsquigarrow \mathit{RecFunc}(\tau_{cpp}, \tau_{cpp}{}', f, x, e_{cpp})}$$

E.2 Traduction des types vers les types C++

Cette section présente les règles de traduction des types SCOL vers C++, utilisés principalement dans la traduction des fonctions récursives.

$$\textsc{TInt}\ \texttt{int} \rightsquigarrow \texttt{int} \qquad\qquad \textsc{TBool}\ \texttt{bool} \rightsquigarrow \texttt{bool}$$

$$\textsc{TList}\ \frac{\tau \rightsquigarrow \tau_{cpp}}{\tau\texttt{list} \rightsquigarrow \texttt{SCOLList} < \tau_{cpp} >}$$

$$\textsc{TTuple}\ \frac{\forall i \in [1,n] \tau^i \rightsquigarrow \tau^i_{cpp}}{\tau^1 * ... * \tau^n \rightsquigarrow \texttt{std} :: \texttt{tuple} < \tau^1_{cpp}, ..., \tau^2_{cpp} >}$$

$$\textsc{TAbs}\ \frac{\tau \rightsquigarrow \tau_{cpp} \qquad \tau' \rightsquigarrow {\tau_{cpp}}'}{\tau \to \tau' \rightsquigarrow \texttt{std} :: \texttt{function} < {\tau_{cpp}}'(\tau_{cpp}) >}$$

$$\textsc{TRow} < row > \rightsquigarrow \texttt{SCOLFunction} \qquad\qquad \textsc{TVar}\ \alpha \rightsquigarrow \texttt{SCOLFunction}$$

$$\textsc{TConstraint}\ \frac{\tau \rightsquigarrow \tau_{cpp}}{\tau\ \texttt{with}\ C_{main} \rightsquigarrow \tau_{cpp}}$$

DOCTORAT BRETAGNE LOIRE MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Contrôle sûr de chaînes d'obfuscation logicielle

Mot clés : Obfuscation, Chaînes de compilation, Langage dédié, Typage, Sémantiques formelles

Résumé :

L'obfuscation de code est une technique de protection de programme qui consiste à rendre la rétro-ingénierie d'un programme plus difficile, afin de protéger des secrets, de la propriété intellectuelle, ou de compliquer la détection (malware). L'obfuscation est généralement effectuée à l'aide d'un ensemble de transformations sur le programme cible. L'obfuscation est également souvent utilisée en conjonction avec d'autres techniques de protection de programmes telles que le *watermarking* ou le *tamperproofing*. Ces transformations sont généralement intégrées dans les transformations de code appliquées pendant le processus de compilation, comme les optimisations. Cependant, appliquer des transformations d'obfuscation demande plus de préconditions sur le code et de précisions dans l'application pour offrir un compromis performance/sécurité acceptable ce qui rend les compilateurs traditionnels peu adaptés à l'obfuscation.

Pour répondre à ces problèmes, cette thèse propose SCOL, un langage qui permet de décrire le processus de compilation d'un programme en prenant en compte les problématiques spécifiques à l'obfuscation. Celui-ci permet de décrire des chaînes de compilation avec un haut niveau d'abstraction, tout en permettant une précision élevée sur la composition des transformations. Le langage possède un système de types qui permet de vérifier la cohérence de la chaîne de compilation. C'est-à-dire que vérifier que la composition des transformations permet d'obtenir le niveau de protection et de couverture visé.

Title: Safe control of obfuscation toolchains

Keywords: Obfuscation, Compilation chains, Domain-Specific Language, Formal semantics, Typing

Abstract: Code obfuscation is a software protection technique that is designed to make reverse engineering a program more challenging, in order to protect secrets, intellectual property, or to complicate detection (malware). The obfuscation is generally carried out using a set of transformations on the target program. Obfuscation is also often used in conjunction with other software protection techniques, such as watermarking or tamperproofing. These transformations are usually integrated into code transformations applied during the compilation process, such as optimisations. However, the application of obfuscation transformations requires more preconditions on the code and more precision in the application to provide an acceptable performance/safety trade-off, making traditional compilers unsuitable for obfuscation.

To address these problems, this thesis presents SCOL, a language that is able to describe the process of compiling a program while considering the specific problems of obfuscation. This allows you to describe compilation chains with a high level of abstraction, while allowing a high degree of accuracy on the composition of the transformations. The language has a type system which checks the correctness of the compilation chain. That means that the composition of transformations provides the targeted level of protection and coverage.