Sécurité du logiciel, sixième cours

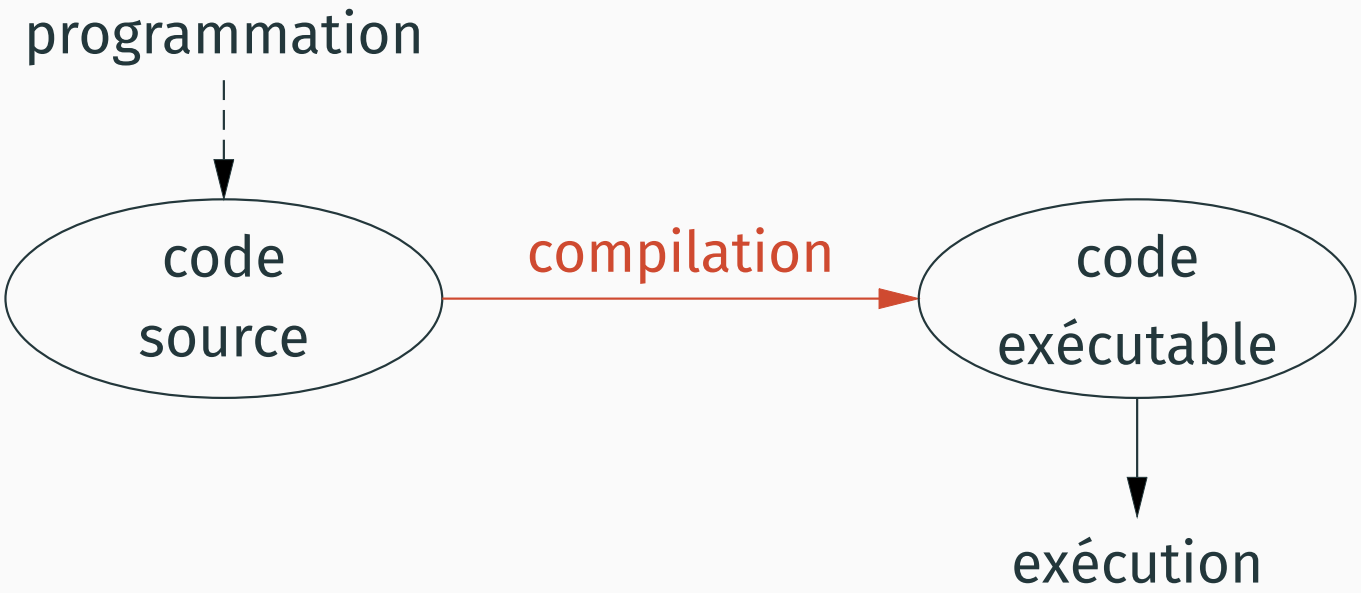
Compilation et sécurité

Xavier Leroy

2022-04-14

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Dans ce cours



Que contribue (positivement ou négativement) la compilation à la sécurité du code compilé (celui qui s'exécute vraiment) ?

Compilation qui met en œuvre des mesures de sécurité

Vérification des bornes lors des accès aux tableaux

Une vérification essentielle pour garantir la sûreté de l'exécution et l'absence d'attaques de type *buffer overflow*.

$$\texttt{x := A[i]} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); x := *(A + i)}$$

$$\texttt{A[i] := x} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); *(A + i) := x}$$

`assert`(b) : arrête l'exécution ou lève une exception si b est faux.

`*(A + i)` représente une lecture ou écriture mémoire directe, sans vérifications.

Vérification des bornes lors des accès aux tableaux

Une vérification essentielle pour garantir la sûreté de l'exécution et l'absence d'attaques de type *buffer overflow*.

$$\texttt{x := A[i]} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); x := *(A + i)}$$

$$\texttt{A[i] := x} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); *(A + i) := x}$$

`size(A)` est la taille du tableau `A`, déterminée

- soit statiquement par la déclaration de `A`, p.ex. `int A[10];`
- soit dynamiquement depuis la représentation de `A`

Éviter les tests de bornes inutiles

Parfois, le contexte de l'accès au tableau garantit que l'accès est dans les bornes :

```
for (int i = 0; i < size(A); i++)  x += A[i];
```

On peut reconnaître ce motif de code et le compiler sans `assert`.

Il est plus sûr et plus général de produire systématiquement le test dynamique (`assert`) puis de l'éliminer si possible, par des optimisations s'appuyant sur des analyses statiques.

(→ cours des 19/12/2019 et 16/01/2020)

Élimination d'assertions redondantes

Pas la peine de tester deux fois de suite la même assertion !
(Un cas particulier de *Common Subexpression Elimination*.)

Code source :

```
t = A[i]; A[i] = t + 1;
```

Ajout des tests de bornes :

```
assert(0 ≤ i < size(A));  t = *(A + i);
assert(0 ≤ i < size(A));  *(A + i) = t + 1;
```

Élimination du test redondant :

```
assert(0 ≤ i < size(A));  t = *(A + i);
~~assert(0 ≤ i < size(A));~~  *(A + i) = t + 1;
```

Optimisation par analyse d'intervalles

L'analyse infère des propriétés de la forme $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: variable du programme, a, b constantes inférées par l'analyse)

Code source :

```
int A[10];
for (int i = 2; i < 8; i++)  {
    x += A[i];
}
```

Optimisation par analyse d'intervalles

L'analyse infère des propriétés de la forme $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: variable du programme, a, b constantes inférées par l'analyse)

Ajout des tests de bornes :

```
int A[10];
for (int i = 2; i < 8; i++)  {
    assert(0 ≤ i < 10);
    x += *(A + i);
}
```

Optimisation par analyse d'intervalles

L'analyse infère des propriétés de la forme $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: variable du programme, a, b constantes inférées par l'analyse)

Analyse d'intervalles :

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(0 ≤ i < 10);
    x += *(A + i);
}
```

Optimisation par analyse d'intervalles

L'analyse infère des propriétés de la forme $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: variable du programme, a, b constantes inférées par l'analyse)

Propagation des informations d'intervalles :

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(true);
    x += *(A + i);
}
```

Optimisation par analyse d'intervalles

L'analyse infère des propriétés de la forme $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: variable du programme, a, b constantes inférées par l'analyse)

Suppression des assertions triviales :

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(true);
    x += *(A + i);
}
```

Pour aller plus loin : analyses relationnelles
(polyèdres $a\mathtt{x} + b\mathtt{y} \leq c$, octogones $\pm\mathtt{x} \pm \mathtt{y} \leq c$, etc)

Déplacement d'assertions hors des boucles

(Un cas particulier de *loop-invariant code motion*.)

Code source :

```
int A[10];
for (int i = 0; i < 10; i++)  {
    B[j] = B[j] + A[i];
}
```

Déplacement d'assertions hors des boucles

(Un cas particulier de *loop-invariant code motion*.)

Ajout des tests de bornes :

```
int A[10];
for (int i = 0; i < 10; i++)  {
    assert(0 ≤ i < 10);
    assert(0 ≤ j < size(B));
    assert(0 ≤ j < size(B));
    *(B + j) = *(B + j) + *(A + i);
}
```

Déplacement d'assertions hors des boucles

(Un cas particulier de *loop-invariant code motion*.)

Application des optimisations précédentes :

```
int A[10];
for (int i = 0; i < 10; i++)  {
    assert(0 ≤ i < 10);
    assert(0 ≤ j < size(B));
    assert(0 ≤ j < size(B));
    *(B + j) = *(B + j) + *(A + i);
}
```

Déplacement d'assertions hors des boucles

(Un cas particulier de *loop-invariant code motion*.)

Déplacement de l'assertion avant la boucle :

```
int A[10];
assert(0 ≤ j < size(B));
for (int i = 0; i < 10; i++)  {
    ~~assert(0 ≤ i < 10);~~
    ~~assert(0 ≤ j < size(B));~~
    ~~assert(0 ≤ j < size(B));~~
    *(B + j) = *(B + j) + *(A + i);
}
```

Expansion en ligne de fonctions

L'expansion en ligne (*inlining*) d'une fonction à son point d'appel permet souvent d'éliminer davantage de tests de bornes.

```
int f(int A[], int i) {
    assert (0 ≤ i < size(A));
    return *(A + i) + 1;
}
int B[10];
int g(void) { return f(B, 2); }
```

Après expansion de `f` dans `g` :

```
int B[10];
int g(void) {
    assert (0 ≤ 2 < size(B));
    return *(B + 2) + 1;
}
```

Protection contre l'attaque Spectre v1

S'assurer que l'exécution spéculative d'un accès hors borne ne va pas accéder «trop loin» du tableau.

$$\texttt{x := A[i]} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{assert(0} \leq \texttt{i} < \texttt{size(A));} \\ \texttt{x := *(A +} \ \texttt{clip(i, size(A))}\texttt{)} \end{array}$$

$$\texttt{A[i] := x} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{assert(0} \leq \texttt{i} < \texttt{size(A));} \\ \texttt{*(A +} \ \texttt{clip(i, size(A))}\texttt{)} \ \texttt{:= x} \end{array}$$

où $\texttt{clip}(i, n)$ est une expression sans branchements telle que

$$\texttt{clip}(i, n) = \begin{cases} i & \text{si } 0 \leq i < n \\ 0 & \textit{sinon} \end{cases}$$

P.ex en x86 : `cmp Ri, Rn; sbb Rc, Rc; and Rc, Ri.`

Autres exemples

Implémentation du typage dynamique.

Protections contre les fautes du matériel.

($\rightarrow$ séminaire K. Heydemann)

Autres attaques sur les exécutions spéculatives.

($\rightarrow$ séminaire F. Piessens)

Obfuscation de code. ($\rightarrow$ séminaire S. Blazy)

Compilation qui supprime des mesures de sécurité

Le contrat du compilateur

Le compilateur produit du code machine, aussi efficace que possible, qui ≪calcule la même chose≫ que le programme source.

Deux hypothèses principales :

- La machine se comporte comme décrit dans le manuel.
  (Pas de fautes, pas d'observations du temps, de l'exécution spéculative, …)
- Le programme source n'a pas de comportements indéfinis.
  (P.ex. parce qu'il a passé le typage statique, ou parce que le programmeur s'y engage.)

≪Optimiser≫ les protections contre les attaques en faute

(Voir le séminaire de K. Heydemann du 7 avril.)

Redondance sur les tests :

```
if (cond) {
    assert (cond);
    ...
} else {
    assert (! cond);
    ...
}
```

Une analyse statique montre trivialement que `cond` est vraie dans la branche `then` et fausse dans la branche `else`

$\rightarrow$ suppression des deux assertions.

≪Optimiser≫ les protections contre les attaques en faute

Redondance entre flux de contrôle et valeurs :

```
t = 0;
if (cond) {
    t |= 1; ...
} else {
    t |= 2; ...
}
assert (t != 0 && t != 3);
```

Une analyse d'intervalles montre que $\texttt{t} \in [1, 2]$ au niveau de l'assertion, et donc elle est toujours vraie et va être supprimée.

≪Optimiser≫ la protection contre Spectre v1

Telle que présentée au 4e cours :

$$x := \texttt{A}[i] \quad \rightsquigarrow \quad \begin{array}{l} \texttt{assert}(0 \leq i < \texttt{size}(\texttt{A})); \\ \quad x := *(\texttt{A} + \texttt{sel}(0 \leq i < \texttt{size}(\texttt{A}), \texttt{ i, } 0)) \end{array}$$

Une analyse statique triviale ≪sait≫ qu'après $\texttt{assert}(b)$ la condition b est vraie. Donc l'accès peut être réécrit en

$$x := *(\texttt{A} + \texttt{sel(true, i, } 0))$$

puis simplifié en

$$x := *(\texttt{A} + \texttt{i})$$

D'où l'importance d'utiliser $*(\texttt{A} + \texttt{clip}(i, \texttt{ size(A)}))$ avec `clip` qui est ≪opaque≫ à l'optimisation.

≪Optimiser≫ les codes en temps constant

Le compilateur a le droit d'utiliser des branchements conditionnels ou des accès mémoire pour implémenter du code source qui n'en contient pas.

Exemple : si `b` est de type `_Bool`,

```
x = b * a1 + (1 - b) * a0;
```

$\rightsquigarrow$ `if (b) x = a1; else x = a0;`

ou encore

$\rightsquigarrow$ `int t[2] = { a0, a1 }; x = t[b];`

≪Optimiser≫ les codes en temps constant (rappel du 4e cours)

(Simon, Chisnall, Anderson, *What you get is what you C : controlling side effects in mainstream C compilers*, 2018).

Expérience : 4 implémentations de `sel` en C portable, compilées pour x86-32 par différentes versions de Clang.

		VERSION_1		VERSION_2		VERSION_3		VERSION_4	
		inlined	library	inlined	library	inlined	library	inlined	library
Clang 3.0	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✗
	-O2	✓	✓	✓	✗	✗	✓	✓	✗
	-O3	✓	✓	✓	✗	✗	✓	✓	✗
Clang 3.3	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗
Clang 3.9	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗

✔ = production de code ≪temps constant≫.

✘ = production d'un branchement conditionnel.

Les comportements indéfinis en C / C++

Le standard ISO C 2018 liste plus de 200 comportements indéfinis possibles…

Certains, comme l'écriture dans un tableau hors des bornes, correspondent à des cas où tout peut effectivement arriver.

Exemple (rappel 1er cours) : débordement de tableau dans la pile

```
int check(void) {
    char b[80]; int ok = 0;
    gets(b); ...; return ok;
}
```

Si `gets` écrit au-delà de `b`, cela peut
– écraser la valeur de `ok` → résultat incorrect
– invalider l'adresse de retour → plantage au retour de `check`
– rediriger l'adresse de retour → exécution de code arbitraire.

Les comportements indéfinis en C / C++

Historiquement, beaucoup de comportements indéfinis sont des cas où différentes machines se comportent différemment, et on veut laisser au compilateur la possibilité de «suivre» le comportement de la machine. P.ex :

- Déréférencement du pointeur `NULL` :
  une instruction `load` ou `store` du processeur peut faire *segmentation fault*, ou accéder normalement à l'adresse 0.
- Débordement dans une addition entière signée :
  l'instruction `add` du processeur peut prendre le résultat modulo 2^N, ou le saturer à `INT_MAX` ou `INT_MIN`, ou arrêter le programme.

Compiler C/C++ en présence de comportements indéfinis

Au XX^{e} siècle : utiliser la liberté laissée par les comportements indéfinis pour traduire les opérations du langage en instructions simples du processeur, tant qu'elles font ce qu'il faut dans les cas définis.

Opération	Code produit
adressage dans un tableau $t + i$	$\texttt{add}(t, \texttt{mul}(i, \texttt{sizeof}(\tau)))$
déréférencement de pointeur $*p$	$\texttt{load}(p)$, $\texttt{store}(p)$
addition entière signée $x + y$	$\texttt{add}(x, y)$

Compiler C/C++ en présence de comportements indéfinis

Au XX^{e} siècle : utiliser la liberté laissée par les comportements indéfinis pour traduire les opérations du langage en instructions simples du processeur, tant qu'elles font ce qu'il faut dans les cas définis.

Au XXI^{e} siècle : utiliser la liberté laissée par les comportements indéfinis pour optimiser en supposant qu'il n'y en a pas, au risque de produire du code absurde s'il y en a.

Opération	Information pour l'optimisation
adressage dans un tableau $t + i$	i est dans les bornes de t
déréférencement pointeur $*p$	$p \neq$ `NULL` et est valide
addition entière signée $x + y$	`INT_MIN` $\leq x + y \leq$ `INT_MAX`

≪Optimiser≫ une division par zéro

(Noyau Linux, `lib/mpi/mpi-pow.c`. Exemple cité par Wang et al, *Undefined behavior : what happened to my code ?*, 2012.)

```
if (!msize)
    msize = 1 / msize; /* provoke a signal */
```

Le compilateur supprime entièrement le test et la division.

```
if (!msize) msize = 1 / msize; else skip;
```

(propagation des constantes)

```
⇝ if (!msize) msize = 1 / 0; else skip;
```

(pas de comportements indéfinis)

```
⇝ if (!msize) unreachable; else skip;
```

(élimination des branchements redondants)

```
⇝ skip
```

≪Optimiser≫ un test de pointeur nul

(Noyau Linux, `drivers/net/tun.c`)

```
unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk; // (1)
    if (!tun) return POLLERR; // (2)
    ...
}
```

Le compilateur supprime la ligne (2) : puisque l'accès `tun->sk` en ligne (1) était défini, forcément `tun` n'est pas `NULL` ...

Permet de monter une attaque en plaçant de la mémoire valide à l'adresse 0 avec `mmap`.

≪Optimiser≫ un test de non-débordement entier

(Noyau Linux, `fs/open.c`; cité par Wang et al.)

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0) return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

Le compilateur fait comme si `offset + len >= 0`, puisque `offset` et `len` sont positifs, et puisque l'addition `offset + len` est supposée ne pas déborder.

≪Optimiser≫ une levée d'exception

Certains compilateurs optimisent en supposant non seulement qu'il n'y a pas eu de comportement indéfini, mais qu'il n'y en aura pas dans le futur !

```
int safe_div(int x, int y)
{
    int res;
    if (y == 0) error("division by zero");  // (1)
    res = x / y;                            // (2)
    return res;
}
```

Le compilateur se permet d'ordonnancer la division (ligne 2) avant le test et l'appel à `error` (ligne 1). Si `error` interrompt l'exécution, cela rend la fonction `safe_div` inutilisable.

Le champ de mines

> *To me, this is deeply dissatisfying, partially because the compiler inevitably ends up getting blamed, but also because it means that huge bodies of C code are land mines just waiting to explode.*
>
> *(Chris Lattner, 2011)*

Un compilateur peut réduire la sécurité d'un code source, notamment en éliminant des mesures de sécurité.

Et pourtant, le compilateur ne fait qu'appliquer

- des optimisations classiques et plutôt naturelles (propagation des constantes, simplification des conditionnelles, ordonnancement des calculs, …);
- le principe ≪comportement indéfini≫ = ≪je peux produire le code machine qui m'arrange le mieux≫.

Que faire ?

Outils d'analyse statique ou dynamique pour détecter les éventuels comportements indéfinis ou optimisations indésirables.

Optimiser moins agressivement.
(Nombreuses options `-fno-*` dans GCC et Clang.)

Définir davantage de comportements.
(P.ex en CompCert C : arithmétique modulo 2^N, ordres d'évaluation, préservation du comportement jusqu'au comportement indéfini.)

Repenser nos choix de langages de programmation ?

Sécurité du code compilé exécuté dans un contexte hostile

Sécurité du code compilé

On voudrait que le code exécutable produit par le compilateur soit aussi sécurisé que le code source, au sens suivant :

> *Toute attaque sur le code exécutable (menant à une violation de l'intégrité, de la confidentialité ou de la disponibilité) peut aussi être menée sur le code source et être expliquée en termes de la sémantique du langage source*.

Cas d'un programme complet exécuté en isolation

Le code source :

- un programme complet, sans variables libres
- interagit via des opérations d'entrée/sortie explicites.

Le code compilé :

- exécuté en isolation ($\rightarrow$ cours du 24 mars)
- seule surface d'attaque : les entrées/sorties.

Préservation sémantique $\Rightarrow$ compilation sécurisée

Un compilateur qui préserve la sémantique des programmes (p.ex. CompCert) garantit que

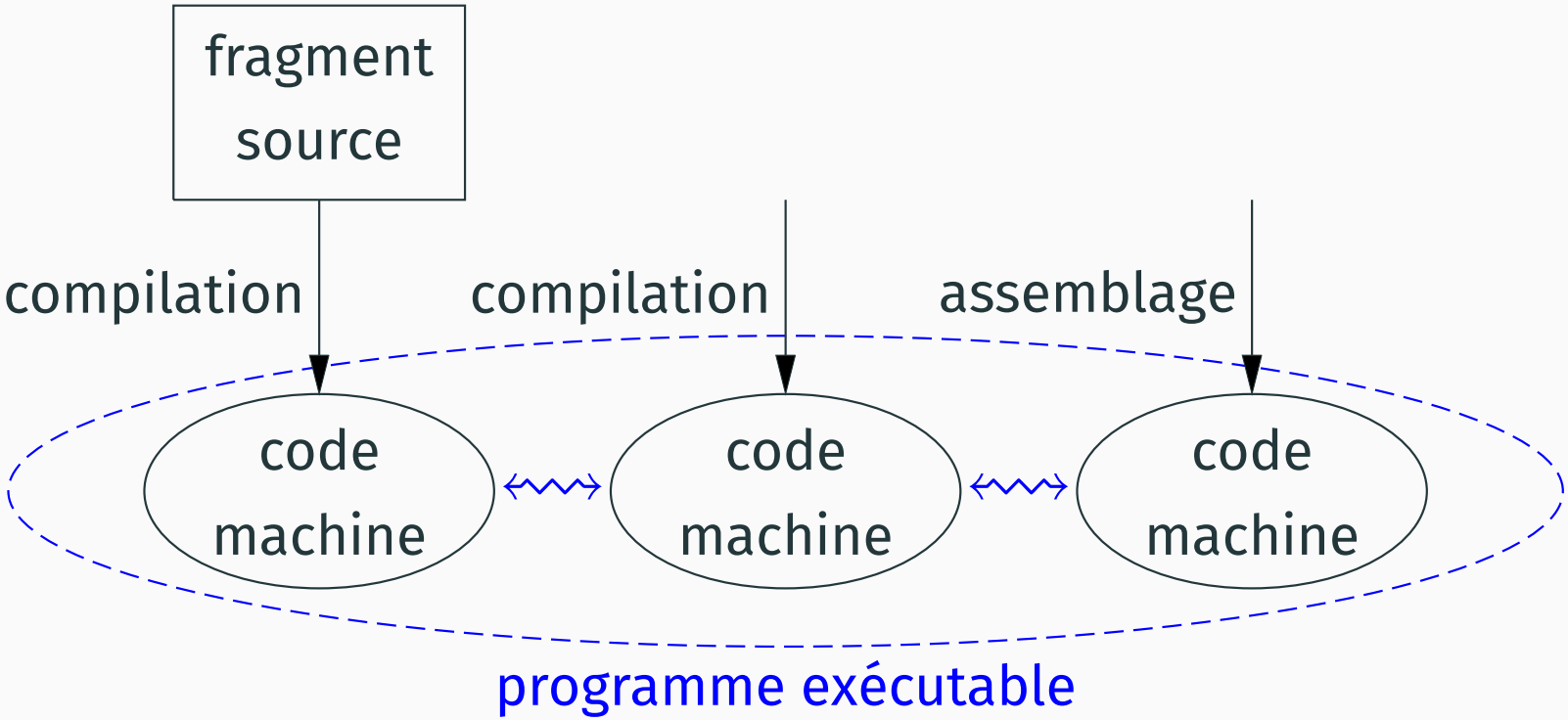
> le comportement du code compilé (p.ex. la trace des E/S) est l'un des comportements permis par la sémantique du code source.

Une attaque sur le code compilé
= une trace T qui provoque un comportement indésirable.

La même trace T provoque le même comportement indésirable dans le code source !

Donc la compilation est sécurisée.

Compilation séparée + édition de liens



Un fragment de programme : fonction, classe, type abstrait, …
compilé en code machine,
puis lié avec d'autres codes machine
(compilés depuis le même langage, ou un autre langage, ou écrits à la main).

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Étant donnée l'adresse p d'une instance de `Account`, un code de niveau machine peut accéder directement au champ `bal` (typiquement à $p + 8$ ou $p + 16$) $\Rightarrow$ *violation de l'invariant.*

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Un code machine peut observer l'état des registres et de la pile au retour d'un appel à `deposit`, et y retrouver la valeur de `bal`.

$\Rightarrow$ *violation de confidentialité.*

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Un code machine peut appeler `deposit` avec un argument `this` qui est le pointeur nul ou un pointeur sur un autre objet

$\Rightarrow$ *plantage ou corruption d'un autre objet.*

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Un code machine peut appeler `deposit` avec `amount` un entier 32 bits qui est plus grand qu'un `short`, trompant ainsi le test de non-débordement $\Rightarrow$ *violation de l'invariant.*

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Un code machine peut sauter directement à l'adresse `withdraw` $+ \delta$, court-circuitant la validation de `amount`

$\Rightarrow$ *violation de l'invariant.*

Attaques niveau ≪machine≫ sur une classe Java

```
class Account {
    private short bal;  // toujours >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Un code machine peut appeler `deposit` avec une adresse de retour et une pile d'appels falsifiées afin de tromper l'inspection de pile du `SecurityManager`. $\Rightarrow$ *attaque ≪confused deputy≫ ?*

Une attaque niveau JVM sur une classe Java

(M. Abadi, *Protection in programming-language translations*, 1998.)

La machine JVM n'a pas de notion de classe interne. Les classes internes de Java sont compilées vers des classes JVM disjointes, ce qui oblige à augmenter la visibilité de champs `private`.

Un source Java :

```
class D {
    class E {
        private int y = x;
    }
    private int x;
    public void set_x(int v) { this.x = v; }
}
```

Une attaque niveau JVM sur une classe Java

(M. Abadi, *Protection in programming-language translations*, 1998.)

La machine JVM n'a pas de notion de classe interne. Les classes internes de Java sont compilées vers des classes JVM disjointes, ce qui oblige à augmenter la visibilité de champs `private`.

Le code JVM compilé, en pseudo-Java :

```
class D {
  private int x;
  public void set_x(int v)
    { this.x = v; }
  static int access$000(D d)
    { return d.x; }
}
```

```
class D$E {
  final D this$0;
  private int y;
  D$E(D d)
  { this$0 = d;
    y = D.access$000(d); }
}
```

Un code JVM écrit à la main peut lire `x` via `D.access$000`.

Attaques ≪bas niveau≫ et équivalences observationnelles

Abadi (1998) propose d'étudier ces attaques ≪bas niveau≫ sur le code compilé en termes d'équivalences observationnelles et de leur non-préservation par la compilation.

Idée grossière :

une attaque ≪bas niveau≫
= deux fragments de code source F_1, F_2
indistinguables au niveau du langage source
dont les codes compilés $\mathcal{C}(F_1)$, $\mathcal{C}(F_2)$
peuvent être distingués au niveau du langage cible.

Exemple de non-préservation d'équivalence observationnelle

```
class D {
    class E {
        private int y = x;
    }
    private int x;
    public void set_x(int v)
    { this.x = v; }
}
```

```
class D {
    class E {
        private int y = 0;
    }
    private int x;
    public void set_x(int v)
    { this.x = v; }
}
```

Les deux classes `D` sont indistinguables par du code Java, mais leurs codes JVM sont distinguables par du code JVM (la classe de gauche a la méthode `access$000` qui révèle `x`, pas celle de droite).

Équivalence observationnelle et *full abstraction*

L'équivalence observationnelle

Deux fragments de code (fonctions, classes, types abstraits, bibliothèques) F_1 et F_2 sont observationnellement équivalents si, pour tout contexte (= programme à trou) C, les programmes complets $C[F_1]$ et $C[F_2]$ se comportent pareil vis-à-vis de la terminaison :

$$F_1 \approx F_2 \quad \stackrel{def}{=} \quad \forall C,\ C[F_1] \text{ termine} \iff C[F_2] \text{ termine}$$

Exemple : les deux classes `D` précédentes sont observationnellement équivalentes.

Exemples d'équivalences observationnelles pour un langage fonctionnel strict typé

Pour les types de base, l'équivalence c'est l'égalité :

$$0 \approx 0 \qquad 0 \not\approx 1$$

(Le contexte $\texttt{if}\ [] = 0\ \texttt{then}\ \Omega\ \texttt{else}\ ()$ distingue 0 et 1.)

Pour les fonctions sur les types de base, l'équivalence est l'égalité extensionnelle (mêmes arguments $\mapsto$ mêmes résultats) :

$$(\lambda x : \texttt{int}.\, x + x) \approx (\lambda x : \texttt{int}.\, x \times 2)$$

$$\texttt{succ} \not\approx \texttt{pred}$$

$$(\lambda x : \texttt{unit}.\, x) \approx (\lambda x : \texttt{unit}.\, ())$$

$$(\lambda x : \texttt{bool}.\, x) \approx (\lambda x : \texttt{bool}.\, \texttt{if}\ x\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false})$$

($\texttt{if}\ []\ 0 = 1\ \texttt{then}\ \Omega\ \texttt{else}\ ()$ distingue $\texttt{succ}$ et $\texttt{pred}$.)

Exemples d'équivalences observationnelles

Pour les fonctions d'ordre supérieur, l'équivalence révèle comment ces fonctions utilisent leurs arguments.

$$(\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,()) \approx (\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,();f\,())$$

$$(\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,()) \not\approx (\lambda f : \texttt{unit} \to \texttt{unit}.\, ())$$

(Le contexte $[\,]\ (\lambda x.\Omega)$ distingue les deux dernières fonctions.)

Le ≪ou séquentiel gauche≫ et le ≪ou séquentiel droit≫ ne sont pas équivalents :

$$\texttt{or}_{\mathit{left}} \stackrel{\mathit{def}}{=} \lambda x, y : \texttt{unit} \to \texttt{bool}.\ \texttt{if}\ x()\ \texttt{then}\ \texttt{true}\ \texttt{else}\ y()$$

$$\texttt{or}_{\mathit{right}} \stackrel{\mathit{def}}{=} \lambda x, y : \texttt{unit} \to \texttt{bool}.\ \texttt{if}\ y()\ \texttt{then}\ \texttt{true}\ \texttt{else}\ x()$$

(Le contexte $[\,]\ (\lambda x.\Omega)\ (\lambda x.\texttt{true})$ les distingue.)

Équivalence observationnelle et propriétés de sécurité

Beaucoup de propriétés basiques de sécurité du langage de programmation se caractérisent par des équivalences observationnelles.

Intégrité des variables locales :

$$\begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in} \\ \quad f(); !x \end{array} \quad \approx \quad \begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad f(); 0 \end{array}$$

Confidentialité des variables locales :

$$\begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = 0\ \mathtt{in}\ f() \end{array} \quad \approx \quad \begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = 1\ \mathtt{in}\ f() \end{array}$$

Équivalence observationnelle et propriétés de sécurité

Indifférence vis-à-vis de l'ordre d'allocation :

$$\begin{array}{l}\texttt{let } x = \texttt{ref } 0 \texttt{ in} \\ \texttt{let } y = \texttt{ref } 0 \texttt{ in} \\ x\end{array} \quad \approx \quad \begin{array}{l}\texttt{let } x = \texttt{ref } 0 \texttt{ in} \\ \texttt{let } y = \texttt{ref } 0 \texttt{ in} \\ y\end{array}$$

Encapsulation procédurale :

$$\begin{array}{l}\texttt{let } c = \texttt{ref } 0 \texttt{ in} \\ \lambda().\, \texttt{incr } c;\ !c\end{array} \quad \approx \quad \begin{array}{l}\texttt{let } c = \texttt{ref } 0 \texttt{ in} \\ \lambda().\, \texttt{decr } c;\ 0 - !c\end{array}$$

Équivalence observationnelle et propriétés de sécurité

Abstraction de types :

```
struct
  type t = permission list
  let init () = [P0;P1;P2]
  let allowed = List.mem
  let drop = List.remove
end : Capa
```

$\approx$

```
struct
  type t = int
  let mask = function
    P0 -> 1 | P1 -> 2 | P2 -> 4
  let init () = 7
  let allowed p c =
    c land mask p <> 0
  let drop p c =
    c land lnot (mask p)
end : Capa
```

(Construire une relation logique est un moyen de montrer l'équivalence observationnelle.)

Équivalence observationnelle et sémantiques dénotationnelles

On considère une sémantique dénotationnelle dans le style de celle de Milner (1978) vue au cours précédent :

$$\mathcal{D} : \mathit{Expr} \to \mathit{Env} \to V$$

où, par exemple, V est un domaine de Scott

$$V = (\mathit{Int} + \cdots + (V \to V) + \{\mathtt{wrong}\})_\perp$$

La sémantique est adéquate si $\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$

La sémantique est complète si $e_1 \approx e_2 \Rightarrow \mathcal{D}(e_1) = \mathcal{D}(e_2)$

La sémantique est *fully abstract* si elle est adéquate et complète.

Adéquation

$$\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$$

Exemple de sémantique non adéquate : une sémantique qui ≪n'a pas assez de valeurs≫ et qui identifie les constantes `true` et `false` du langage. Mais c'est une mauvaise sémantique !

En pratique, les ≪bonnes≫ sémantiques dénotationnelles sont toujours adéquates.

Adéquation

$$\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$$

Esquisse de démonstration :

Supposons $e_1 \not\approx e_2$. On a donc un contexte C tel que

$$C[e_1] \text{ diverge} \qquad C[e_2] \text{ termine}$$

On s'attend à ce que la sémantique distingue les termes qui divergent de ceux qui terminent :

$$\mathcal{D}(C[e_1]) = \bot \qquad \mathcal{D}(C[e_2]) \neq \bot$$

On s'attend à ce que la sémantique soit compositionnelle : $\mathcal{D}(C[e])$ est une fonction de $\mathcal{D}(e)$.

D'où $\mathcal{D}(e_1) \neq \mathcal{D}(e_2)$, et le résultat, par contraposée.

Complétude

$$e_1 \approx e_2 \Rightarrow \mathcal{D}(e_1) = \mathcal{D}(e_2)$$

Beaucoup de ≪bonnes≫ sémantiques sont incomplètes, typiquement parce que le domaine V contient des valeurs sémantiques non définissables dans le langage de programmation.

Exemple : dans les domaines de Scott, les fonctions continues $V \to V$ contiennent des fonctions ≪parallèles≫ qui ne sont pas définissables dans un langage purement séquentiel.

Le «ou parallèle»

Une fonction «ou» paresseuse qui renvoie `true` dès que l'un de ses arguments renvoie `true`.

$$\texttt{por}\ x\ y = \begin{cases} \texttt{true} & \text{si } x() = \texttt{true} \text{ (même si } y() \text{ diverge)} \quad (1) \\ \texttt{true} & \text{si } y() = \texttt{true} \text{ (même si } x() \text{ diverge)} \quad (2) \\ \texttt{false} & \text{si } x() = \texttt{false} \text{ et } y() = \texttt{false} \\ \bot & \text{sinon} \end{cases}$$

Elle n'est pas définissable dans un langage séquentiel : le calcul doit ou bien commencer par évaluer $x()$, ce qui invalide (2), ou bien commencer par évaluer $y()$, ce qui invalide (1).

Incomplétude des domaines de Scott

(G. Plotkin, *LCF considered as a programming language*, 1977.)

On définit deux ≪goûteurs de "ou"≫ M_0 et M_1 :

$$M_k \overset{def}{=} \lambda f : (\mathtt{unit} \to \mathtt{bool}) \to (\mathtt{unit} \to \mathtt{bool}) \to \mathtt{bool}.$$
$$\mathtt{if}\ f\,(\lambda\_.\mathtt{true})\,(\lambda\_.\Omega)$$
$$\wedge\, f\,(\lambda\_.\Omega)\,(\lambda\_.\mathtt{true})$$
$$\wedge\, \neg\, f\,(\lambda\_.\mathtt{false})\,(\lambda\_.\mathtt{false})$$
$$\mathtt{then}\ k\ \mathtt{else}\ \Omega$$

M_0 et M_1 n'ont pas la même dénotation, car

$$\mathcal{D}(M_0)\,\mathtt{por} = 0 \neq 1 = \mathcal{D}(M_1)\,\mathtt{por}$$

Et pourtant $M_0 \approx M_1$ car aucune fonction f définissable dans le langage ne satisfait les 3 conditions du `if`.

Compilation *fully abstract*

Compilation *fully abstract*

Abadi (1998) définit la *full abstraction* pour un compilateur comme étant la préservation des équivalences observationnelles dans les deux directions :

$$F_1 \approx_{\text{source}} F_2 \quad \text{si et seulement si} \quad \mathcal{C}(F_1) \approx_{\text{compilé}} \mathcal{C}(F_2)$$

Un compilateur *fully abstract* est intéressant pour la sécurité :

- Toutes les attaques ≪bas niveau≫ (par un contexte écrit à la main en langage machine) sont aussi possibles au ≪haut niveau≫ (par un contexte écrit en langage source puis compilé).
- On peut donc raisonner sur la correction et la sécurité d'un programme entièrement au niveau du langage source.

≪Adéquation≫ : compilation préservant la sémantique

$$F_1 \not\approx_{\text{source}} F_2 \quad \Longrightarrow \quad \mathcal{C}(F_1) \not\approx_{\text{compilé}} \mathcal{C}(F_2)$$

C'est généralement une conséquence du fait que le compilateur préserve la sémantique des programmes source.

Soit S un contexte source tel que $S[F_1]$ diverge mais pas $S[F_2]$.

Par préservation de la sémantique, le code compilé $\mathcal{C}(S[F_1])$ diverge mais pas le code compilé $\mathcal{C}(S[F_2])$.

Si le compilateur est compatible avec la compilation séparée,
$\mathcal{C}(S[F_1]) = M[\mathcal{C}(F_1)]$ et $\mathcal{C}(S[F_2]) = M[\mathcal{C}(F_2)]$,
où M est un contexte machine obtenu par traduction de S.

Donc $\mathcal{C}(F_1)$ et $\mathcal{C}(F_2)$ ne sont pas observationnellement équivalents.

≪Complétude≫ : compilation préservant les équivalences

$$F_1 \approx_{\text{source}} F_2 \quad \Longrightarrow \quad \mathcal{C}(F_1) \approx_{\text{compilé}} \mathcal{C}(F_2)$$

C'est la partie difficile de la compilation *fully abstract*.

On a vu de nombreux exemples où un contexte M de niveau ≪machine≫ peut, par inspection directe de la mémoire et des registres, distinguer les codes compilés $\mathcal{C}(F_1)$ et $\mathcal{C}(F_2)$.

Il est souvent impossible de construire un contexte source S par ≪traduction arrière≫ de M qui distinguerait F_1 et F_2 au niveau du langage source.

Un exemple de compilation *fully abstract* : d'un langage statiquement typé à un langage dynamiquement typé

Langage source : un λ-calcul simplement typé, avec des booléens, des produits, et des sommes.

Types : $\tau, \sigma ::= \texttt{unit} \mid \texttt{bool} \mid \sigma \to \tau$
$\mid \sigma \times \tau \mid \sigma + \tau$

Termes :	$a ::= x \mid \lambda x : \tau.\, a \mid a_1\, a_2 \mid \texttt{fix}$	fonctions (récursives)
	$\mid \texttt{()} \mid \texttt{false} \mid \texttt{true}$	
	$\mid \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3$	booléens
	$\mid (a_1, a_2) \mid \texttt{fst}(a) \mid \texttt{snd}(a)$	produits
	$\mid \texttt{inl}(a) \mid \texttt{inr}(a) \mid \texttt{match}\ldots$	sommes

Langage cible : le même λ-calcul, mais dynamiquement typé.

Compilation

(Devriese, Patrignani, Piessens, *Fully-abstract compilation by approximate back-translation*, 2016.)

Compilation de base : effacer les types : $\mathcal{C}(a) = \overline{a}$

Mais cela ne préserve pas les équivalences !

$$\lambda x : \mathtt{unit}.\ x \quad \approx_{\text{typée}} \quad \lambda x : \mathtt{unit}.\ ()$$
$$\lambda x.\ x \quad \not\approx_{\text{non typée}} \quad \lambda x.\ ()$$

Compilation sécurisée : effacer les types + protéger le terme par une vérification dynamique des valeurs provenant du contexte.

$$\mathcal{C}(a : \tau) = \mathtt{protect}_{\tau}(\overline{a})$$

Protection contre les contextes mal typés

Du monde typé vers le monde non typé :

$$\texttt{protect}_{\texttt{unit}} = \lambda x.x$$
$$\texttt{protect}_{\texttt{bool}} = \lambda x.x$$
$$\texttt{protect}_{\sigma\times\tau} = \lambda x.(\texttt{protect}_{\sigma}(\texttt{fst}(x)), \texttt{protect}_{\tau}(\texttt{snd}(x)))$$
$$\texttt{protect}_{\sigma\to\tau} = \lambda f.\lambda x.(\texttt{protect}_{\tau}(f\ (\texttt{confine}_{\sigma}(x))))$$

Du monde non typé au monde typé :

$$\texttt{confine}_{\texttt{unit}} = \lambda x.()$$
$$\texttt{confine}_{\texttt{bool}} = \lambda x.\ \texttt{if}\ x\ \texttt{then true else false}$$
$$\texttt{confine}_{\sigma\times\tau} = \lambda x.(\texttt{confine}_{\sigma}(\texttt{fst}(x)), \texttt{confine}_{\tau}(\texttt{snd}(x)))$$
$$\texttt{confine}_{\sigma\to\tau} = \lambda f.\lambda x.(\texttt{confine}_{\tau}(f\ (\texttt{protect}_{\sigma}(x))))$$

Exemple de protection

$$
\begin{aligned}
&\mathcal{C}(\lambda x : \texttt{bool}.\, x) \\
&\quad = \quad \lambda x.\, \texttt{let } x = \texttt{if } x \texttt{ then true else false in } x \\
&\mathcal{C}(\lambda x : \texttt{bool}.\, \texttt{if } x \texttt{ then true else false}) \\
&\quad = \quad \lambda x.\, \texttt{let } x = \texttt{if } x \texttt{ then true else false in} \\
&\qquad\qquad \texttt{if } x \texttt{ then true else false}
\end{aligned}
$$

On a bien équivalence observationnelle même dans des contextes non typés : les deux fonctions protégées envoient `true` vers `true`, `false` vers `false`, et les non-booléens vers `wrong`.

Full abstraction par traduction inverse de contextes

Soit M un contexte non typé qui distingue $\mathcal{C}(a_1 : \tau)$ et $\mathcal{C}(a_2 : \tau)$

$$M\ (\texttt{protect}_\tau(\overline{a_1}))\ \text{diverge} \qquad M\ (\texttt{protect}_\tau(\overline{a_2}))\ \text{termine}$$

Peut-on traduire M en un contexte typé S qui distingue a_1 et a_2 ?

Oui, assez facilement, si le langage source contient un type universel `U` pour représenter les termes non typés :

```
type U = Wrong | Unit | Bool of bool
       | Pair of U * U | Fun of (U -> U)
```

On traduit M en un contexte typé S_U avec trou de type `U`, puis en S avec trou de type τ.

Full abstraction par traduction inverse de contextes

Soit M un contexte non typé qui distingue $\mathcal{C}(a_1 : \tau)$ et $\mathcal{C}(a_2 : \tau)$

$$M\ (\texttt{protect}_\tau(\overline{a_1}))\ \text{diverge} \qquad M\ (\texttt{protect}_\tau(\overline{a_2}))\ \text{termine}$$

Peut-on traduire M en un contexte typé S qui distingue a_1 et a_2 ?

Oui, plus difficilement, si le langage source n'a pas de types récursifs, en utilisant des approximations de niveau n du type universel :

$$U_0 = \texttt{unit} \qquad U_{n+1} = \texttt{unit} + \texttt{bool} + (U_n \times U_n) + (U_n \to U_n)$$

n est choisi assez grand pour ne pas tomber à 0 avant d'avoir observé la différence de comportement entre a_1 et a_2.

(Devriese, Patrignani, Piessens, *Fully-abstract compilation by approximate back-translation*, 2016.)

Compilation *fully abstract* d'un langage statiquement typé vers JavaScript

(Fournet, Swamy, Chen, Dagand, Strub, Livshits, *Fully Abstract Compilation to JavaScript*, 2013.)

Langage source : fonctionnel et impératif, statiquement typé (initialement un fragment de F*; ultérieurement, TypeScript).

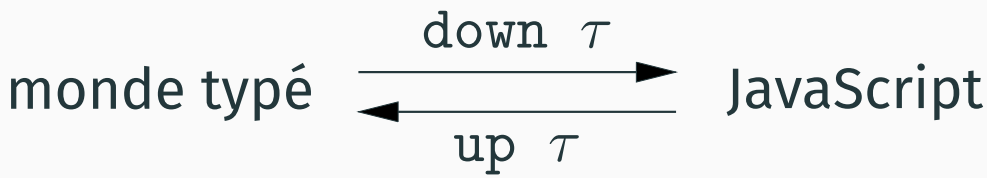
Langage cible : JavaScript.

Compilation : traduction naïve qui efface les types

$$[\![\lambda x : \tau.\ a]\!] = \texttt{function}(x)\{\texttt{var}\ \vec{y}; \texttt{return}\ [\![a]\!];\}$$
$$[\![a_1\ a_2]\!] = [\![a_1]\!]\ [\![a_2]\!]$$
$$[\![(a_1, a_2)]\!] = \{\texttt{tag} : \texttt{"Pair"};\ 0 : [\![a_1]\!];\ 1 : [\![a_2]\!]\}$$

puis protection vis-à-vis du contexte.

Fonctions de protection




```
function downunit(x) { return x;}
function upunit(x) { return undefined;}
function downbool(x) { return x;}
function upbool(z) { return (z ? true : false);}
function downstring(x) { return x;}
function upstring(x) { return (x + "");}

function downpair(dn_a, dn_b) {
  return function (p) {
    return {"tag":"Pair", "0":dn_a(p["0"]),"1":dn_b(p["1"])};}}
function uppair(up_a, up_b) {
  return function(z) {
    return {"tag":"Pair", "0":up_a(z["0"]),"1":up_b(z["1"])};}}
```

Protection des fonctions

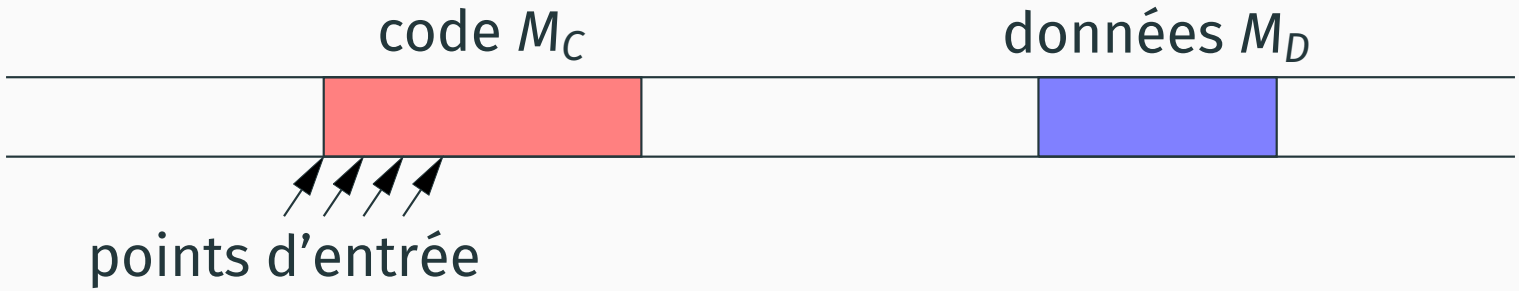
```
function downfun (up_a,down_b) {
  return function (f) {
    return function (z) { return (down_b (f (up_a(z)))); }}}

function upfun (down_a,up_b) {
  return function (f) {
    return function (x) {
      var z = down_a(x);
      var y = undefined;
      function stub(b) {
        if (b) { stub(false); } else { y = up_b(f(z)); } }
      stub(true); return y; };};}
```

Diverses astuces pour empêcher `f` (qui est fournie par le contexte JavaScript, possiblement malveillant) d'inspecter la pile d'appels et d'interférer avec la protection.

Utilisation de protections matérielles de type ≪enclave≫

(Agten, Strackx, Jacobs, Piessens, *Secure compilation to modern processors*, 2012).

On suppose deux zones mémoire protégées, une zone de code M_C et une zone de données M_D :



Le code hors de M_C peut uniquement sauter à un des points d'entrée.

Seul le code dans M_C peut accéder aux données M_D ou sauter ailleurs dans M_C.

(Exemples : enclaves Intel SGX ; *Memory Access Controllers* des cartes à puce.)

Langage source (simplifié)

Des objets alloués statiquement, avec des variables d'instance privées et des méthodes privées ou publiques, avec arguments et résultats de types de base.

```
class Account {
    private short bal = 0;
    public void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    public short balance() { return bal; }
}
```

Compilation (simplifiée)

Les variables d'instance et la pile privée sont placées en zone de donnée protégée M_D.

Le code est placé en zone de code protégée M_C.

Les méthodes publiques sont des points d'entrée.

À l'entrée de chaque méthode publique :

- commutation sur la pile privée et sauvegarde adresse retour
- validation des arguments (`short` est dans $[-2^{15}, 2^{15} - 1]$).

À la sortie :

- effacement des registres autres que le registre résultat
- récupération adresse retour R et validation $R \notin M_C$
- commutation sur la pile d'origine et saut à l'adresse R.

Principaux résultats

(Agten, Strackx, Jacobs, Piessens, *Secure compilation to modern processors*, 2012 ; Patrignani, Clarke, Piessens, *Secure compilation of object-oriented components to protected module architectures*, 2013.)

Extensions du schéma de compilation :

- *Callbacks* depuis les méthodes vers des fonctions (en zone de code non protégée) passées en arguments.
- Allocation dynamique d'objets dans la mémoire protégée.

Des sémantiques à base de traces d'exécutions, capturant les appels et retours de fonctions et de méthodes.

Un résultat de *full abstraction* : si un contexte machine M en zone non protégée distingue $\mathcal{C}(O_1)$ et $\mathcal{C}(O_2)$, on sait construire un contexte source S qui distingue O_1 et O_2 à partir des traces de $M[\mathcal{C}(O_1)]$ et $M[\mathcal{C}(O_2)]$.

Point d’étape

Principes de la compilation sécurisée

Un domaine de recherche actif.

Une caractérisation en termes de *full abstraction*, élégante, extraordinairement difficile à réaliser.

D'autres caractérisations ont été proposées, en termes de compilation qui préserve les (hyper-)propriétés :

- propriétés d'une exécution du programme
  $\rightarrow$ préservation des garanties d'intégrité
- hyper-propriétés reliant deux exécutions du programme
  $\rightarrow$ préservation des garanties de confidentialité.

(Voir le survey de Patrignani, Ahmed et Clarke, *Formal Approaches to Secure Compilation*, 2019.)

Pratique de la compilation sécurisée

Une tension (de nature sociale) entre

- l'optimisation agressive de codes (perçus comme) sans enjeux sécuritaires ;
- le respect des intentions du code source, incluant les protections de sécurité.

Un choix pas évident entre

- ajouter les protections de sécurité pendant la compilation,
- ou les ajouter au niveau source et les préserver pendant la compilation.