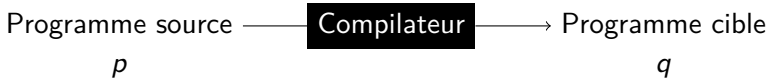
Modèle de fuite structurée & applications à la compilation sécurisée

Vincent Laporte
avec Gilles Barthe Benjamin Grégoire Swarn Priya

GT LVP – GdR GPL – Journées AFADL 2021 (16 juin)

Du bon usage d'un compilateur correct



Correction

$$\forall \sigma\ \sigma', q : \sigma \Downarrow \sigma' \implies p : \sigma \Downarrow \sigma'$$

Il suffit de raisonner sur les comportements de p pour prouver des résultats sur q

- Sans examiner q
- Sans examiner le compilateur
- (Si le langage est déterministe, il suffit de prouver la réciproque)

Limite

Et si ce qui nous intéresse n'est pas décrit par la sémantique ?

Sémantique instrumentée, exemple

On décore le jugement sémantique

La sémantique est enrichie d'une *observation* κ:

$$p : \sigma \Downarrow_{\kappa} \sigma'$$

Exemple : compteurs d'instructions

Pour chaque point de programme, on compte combien de fois l'exécution l'a visité
On peut définir de nouvelles notions, comme le « temps » d'exécution :

$$T(\kappa) = \sum_{i \in p} \kappa[i].$$

Et raisonner sur ces notions :

$$\exists c_0\ c_1, \forall \sigma\ \kappa\ \sigma', p : \sigma \Downarrow_{\kappa} \sigma' \implies T(\kappa) \leqslant c_0 + c_1 \cdot |\sigma[m]|.$$

Sémantique instrumentée pour la sécurité

Fuites par canaux auxiliaires

- La mémoire cache est partagée
- Modélisé par une fuite : adresses, branchements conditionnels

Exemple : « constant-time »

Ce qui fuit vers l'adversaire (λ_1, λ_2) ne dépend pas des secrets (caractérisés par $\equiv$):

$$\forall \sigma_1\ \sigma_2\ \lambda_1\ \lambda_2\ \sigma_1'\ \sigma_2',$$
$$\sigma_1 \equiv \sigma_2 \implies p : \sigma_1 \Downarrow_{\lambda_1} \sigma_1' \implies p : \sigma_2 \Downarrow_{\lambda_2} \sigma_2' \implies \lambda_1 = \lambda_2.$$

Correction instrumentée

Extension naïve de la propriété de correction

$$\forall \sigma\ o\ \sigma', p : \sigma \Downarrow_o \sigma' \implies q : \sigma \Downarrow_o \sigma'.$$

- Faux en général : le compilateur *optimise*

Compilateur instrumenté

- Le compilateur produit un programme cible *et* un **transformateur de fuite** F :

$$C(p) = (q, F).$$

- Le théorème suivant est souvent prouvable :

$$\forall \sigma\ o\ \sigma', p : \sigma \Downarrow_o \sigma' \implies q : \sigma \Downarrow_{F(o)} \sigma'.$$

Jasmin,[1] pour l'implémentation de primitives cryptographiques de qualité

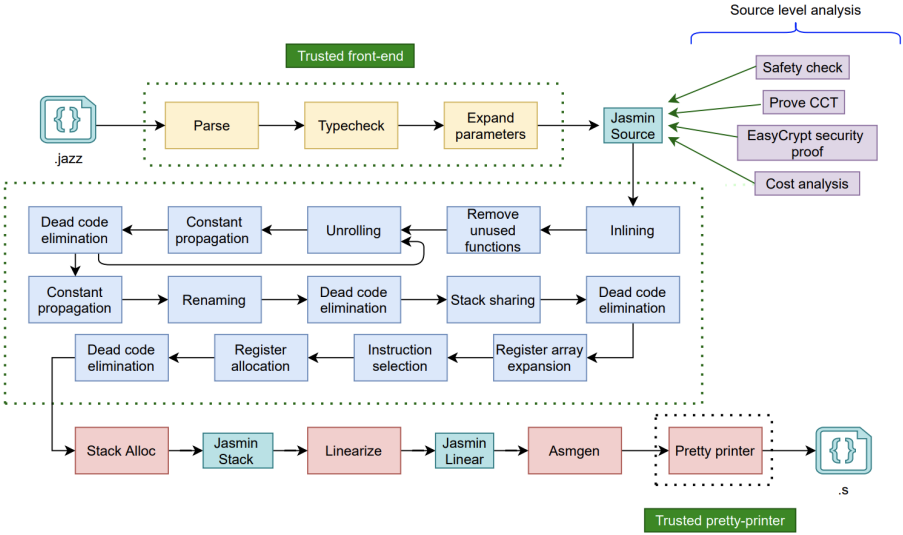
But: allier correction, performance, et sécurité de l'implémentation

1. Un **langage** alliant abstractions (haut niveau) et contrôle (bas niveau)
2. Un **compilateur** prédictible dont la correction est formellement prouvée en Coq
3. Des **outils de vérification** automatiques ou interactifs

Cas d'étude majeurs : Curve25519, SHA3, Chacha/Poly1305

[1] https://github.com/jasmin-lang/jasmin

Le compilateur Jasmin



Aperçu (extrait d'une implémentation de référence de XXH3)

```
inline fn mix2accs(stack u64[8] acc, inline int off, reg u64 secret) −→ reg u64 {
    reg u64[2] data;
    reg u64 m;
    inline int i;
    for i = 0 to 2 {
        data[i] = acc[i + off];
        data[i] ^= [secret + 8 * i];
    }
    m = mul128_fold64(data);
    return m;
}
```

Fuite structurée

Objectifs

- Modéliser des caractéristiques non-fonctionnelles :
  - temps de calcul
  - utilisation de la mémoire cache
- Permettre de décrire comment la compilation transforme la fuite

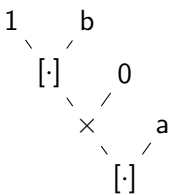
Idées

- La fuite est structurée comme l'exécution
- Les transformations sont exprimées dans un langage dédié qui contient :
  - des transformations structurelles génériques
  - des transformations spécifiques adaptées à chaque passe de compilation

Des arbres pour les expressions

- Exemple : a[b[1] × 0]

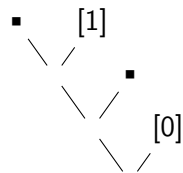
Syntaxe :



Sémantique :

$$\dfrac{\dfrac{1 \downarrow^{\sigma} 1 \qquad \sigma(b,1) = 2}{b[1] \downarrow^{\sigma} 2} \qquad 0 \downarrow^{\sigma} 0}{b[1] \times 0 \downarrow^{\sigma} 0} \qquad \sigma(a,0) = 3}{a[b[1] \times 0] \downarrow^{\sigma} 3}$$

Fuite :



On note l'évaluation instrumentée des expressions ainsi : $e \downarrow^{\sigma}_{\lambda} v$.

Fuites structurées : exécution des instructions

Syntaxe	Sémantique	Fuite
$\{i; c\}$	$\dfrac{i : \sigma \Downarrow_{\lambda_i} \sigma_i \qquad \{c\} : \sigma_i \Downarrow_{\lambda_c} \sigma'}{\{i; c\} : \sigma \Downarrow \sigma'}$	$\{\lambda_i; \lambda_c\}$
$d := e$	$\dfrac{e \downarrow^{\sigma}_{\lambda_e} v \qquad d := v \downarrow^{\sigma}_{\lambda_d} \sigma'}{d := e : \sigma \Downarrow \sigma'}$	$\lambda_d := \lambda_e$
if e then $c_\top$ else $c_\bot$	$\dfrac{e \downarrow^{\sigma}_{\lambda_e} b \qquad c_b : \sigma \Downarrow_{\lambda} \sigma'}{\text{if } e \text{ then } c_\top \text{ else } c_\bot : \sigma \Downarrow \sigma'}$	$\text{if}_b(\lambda_e, \lambda)$

Transformations de fuites d'expressions

τ_e	$::=$	$\blacksquare$	suppression
	$\mid$	id	identité
	$\mid$	π_i	projection
	$\mid$	rep_n	réplication
	$\mid$	$(\tau_e, \ldots, \tau_e)$	composition parallèle
	$\mid$	$\tau_e \circ \tau_e$	composition
	$\mid$	$\ldots$	

Exemples

$$\begin{array}{ccc} 0 \times e & \xrightarrow[\blacksquare]{} & 0 \\ (\blacksquare, \lambda) & & \blacksquare \end{array}$$

$$\begin{array}{ccc} 0 + e & \xrightarrow[\pi_2]{} & e \\ (\blacksquare, \lambda) & & \lambda \end{array}$$

$$\begin{array}{ccc} e_1 + e_2 & \xrightarrow[(\tau^1, \tau^2)]{} & e_1' + e_2' \\ (\lambda_1, \lambda_2) & & (\lambda_1', \lambda_2') \end{array}$$

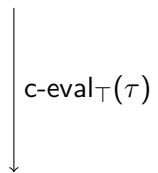
$$\begin{array}{ccc} e_1 + e_2 & \xrightarrow[\tau^2 \circ \pi_2]{} & e_2' \\ (\lambda_1, \lambda_2) & & \lambda_2' \end{array}$$

Transformations de fuites d'instructions

τ	::=	$\tau_e := \tau_e$	affectation
	|	$\text{if}(\tau_e, \tau, \tau)$	conditionnelle
	|	$\text{while}(\tau_e, \tau)$	boucle
	|	$\tau; \tau$	séquence
	|	remove	suppression
	|	$\text{c-eval}_b(\tau)$	évaluation d'une condition
	|	$\ldots$	

Exemple

if e
then $c_\top$ $\quad\quad$ $\text{if}_\top(\lambda_e, \lambda)$
else $c_\bot$



Préservation des contre-mesures aux fuites par canaux auxiliaires

Instrumentation du compilateur Jasmin

- Chaque passe C_i est instrumentée: $C_i(p_i) = (p_{i+1}, F_i)$
- Le compilateur est une composition de passes:

$$(C_{n-1} \circ \cdots \circ C_0)(p_0) = (p_n, F_{n-1} \circ \cdots \circ F_0).$$

- Chaque théorème de correction est étendu aux sémantiques instrumentées

Corollaire : la compilation préserve *constant-time*

- Par construction, les transformateurs de fuite ne dépendent pas de données sensibles
- Leur existence suffit pour conclure

Transformation de coût

- Le coût d'une exécution $p : \sigma \Downarrow_\ell \sigma'$ est $\kappa(\ell)$
- Un transformateur de fuite F induit un transformateur de coût $[\![F]\!]_\kappa$
- Sa *correction* s'énonce :

$$\kappa(F(\ell)) \sqsubseteq [\![F]\!]_\kappa(\kappa(\ell)).$$

Exemple

- κ: compteur de visite pour chaque bloc de base (source) ou instruction (cible)
- $[\![F]\!]_\kappa$ fait correspondre les points du programme cible aux blocs de base du source
- On prouve l'exactitude pour toutes les passes sauf une (déroulage de boucles)

Coût après compilation exprimé au niveau source

Analyse de coût au niveau source

- Par interprétation abstraite, on infère des relations linéaires entre
  - les compteurs de visite de chaque bloc de base
  - les entrées du programme source
- C'est une simple variation de l'analyse de sûreté

Interprétation du résultat d'analyse au niveau cible

- Le *transformateur de coût* permet de déduire des relations linéaires entre
  - les compteurs de visite de chaque instruction
  - les entrées du programme cible
- Résolution: en fixant la valeur de certaines entrées (p.ex.: longueur du message), on en déduit le nombre d'instructions exécutées

Conclusions

Résumé

- En utilisant un modèle de fuite structurée, on peut précisément décrire comment un compilateur transforme ces fuites
- Cela permet de prouver la préservation de contre-mesures (p.ex. constant-time)
- Cela permet de raisonner au niveau source quant au coût du programme compilé

Prolongements

- Considérer des modèles d'adversaire plus sophistiqués
  - temps d'exécution variable pour la division
  - exécution spéculative (Spectre)