Formalisation des garanties de sécurité apportées par l'isolation de composants logiciels

Stage de fin de DUT Informatique

Eng Boris

IUT de Montreuil (Paris 8)
Département Informatique
Promotion 2015-2016

Superviseurs : Yannis Juglaret (Inria), Rémi Georges (Paris 8)

Introduction

Where

- *Inria de Paris* Research center (12 weeks)
- Prosecco team

What

- Research project : computer security & programming languages theory
- Mathematical proofs of properties and implementation with the *Coq* proof assistant

Présentation de Inria

Inria

INVENTEURS DU MONDE NUMÉRIQUE

Institut national français de recherche en informatique et mathématiques.

- 8 centres de recherche
- **Applications** : informatique pure, simulation, robotique, santé, biologie...

Prosecco

PROSECCO
PROGRAMMING SECURELY WITH CRYPTOGRAPHY

- Inria de Paris
- **Activités** : mathématiques pour la sécurité informatique
- **Supervision** : *Yannis Juglaret*
  - Doctorant
  - Sécurité matérielle et compilation sécurisée.

Contexte du projet

Beyond Good and Evil : problématique

- **Beyond Good and Evil (2016)** : Yannis Juglaret, Cătălin Hriţcu *et al.*

Problématique : langage C

- Programme C avec tableau de 3 cases

...	0	1	2	–	...
...	Donnée	Donnée	Donnée	Code	...

Beyond Good and Evil : problématique

- **Beyond Good and Evil (2016)** : Yannis Juglaret, Cătălin Hriţcu *et al.*

Problématique : langage C

- Mauvaise intention → Injection de code

...	0	1	2	–	...
...	Code	Code	Code	Code	...

Buffer Overflow ☹

Beyond Good and Evil : problématique

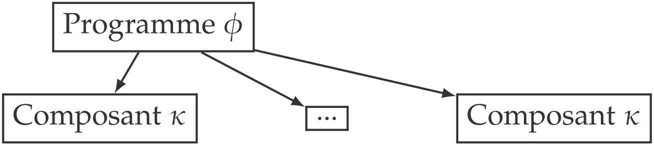
Accès hors borne = **comportement indéfini**.

Langage C :

- Pas de vérifications → ralentissements
- On utilise le C pour ses performances

Beyond Good and Evil : proposition

- Mécanisme de compartimentation



Exemple : navigateurs web

- **Propriété** : Compilation Compartimentée Sécurisée
  - Formalise les garanties de sécurité de la compartimentation
  - Formalise le modèle de l'attaquant

Rôle joué

Assistant de preuve Coq

```
Require Import Induction.

Theorem plus_0_r :
  forall (n:nat), n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
  reflexivity.
Qed.
```

```
2 subgoals
______________________________________(1/2)
0 + 0 = 0
______________________________________(2/2)
S n' + 0 = S n'
```

- Programmation fonctionnelle
- Preuve ↔ Programme (Curry-Howard)
- Utilisation : Mathématiques/Génie logiciel

Objectif

Section 4 de l'article : instance de langage

- **Source** : langage C
- **Cible** : assembleur
- **Compilateur**

Travail effectué :

- Représentation des concepts
- Preuves de propriétés
- Transposition du théorème final

Modélisation des langages

Langage source

- Langage impératif simple
  - Buffers, appels de procédures
- Unique type : entier
- **Exemple** :

component 0 {
  buffer0 = {0, 0, ...};
  buffer1 = {1, 2, 3};
  procs0 = { code };
  procs1 = { code };
  ...
}
	component 1 {
	  buffer0 = {1, 2, ...};
	  buffer1 = {5, 5, 5};
	  procs0 = { code };
	  procs1 = { code };
	  ...
	}

Langage source : syntaxe & sémantique

- Syntaxe

$$e ::= i \mid e_1 \otimes e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid b[e] \mid b[e_1] := e_2 \mid C.P(e) \mid exit$$

- Sémantique opérationnelle

$\mathcal{R}$_If_Vrai :=

$$i \neq 0 \vdash (\text{if } i \text{ then } e_1 \text{ else } e_2) \rightarrow e_1$$

Langage cible

- Jeu d'instructions
  - Nop
  - $Const\ i \rightarrow r$
  - $Mov\ r_1 \rightarrow r_2$
  - $BinOp\ r_1 \otimes r_2 \rightarrow r_3$
  - $Load\ *r_1 \rightarrow r_2$
  - $Store\ *r_1 \leftarrow r_2$
  - $Jal\ r$
  - $Jump\ r$
  - $Call\ C\ P$
  - $Return$
  - $Bnz\ r\ i$
  - $Halt$

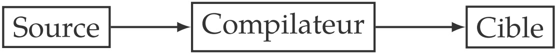
Mémoire

Adresse	0	1	2	...
Donnée	...	...	...	...

Registres

Identifiant	r_{pc}	r_{sp}	...
Contenu	...	...	...

Compilateur



- Fonction de compilation ($\lambda \downarrow$)
  - Correspondance expressions-instructions
  - Organise la mémoire

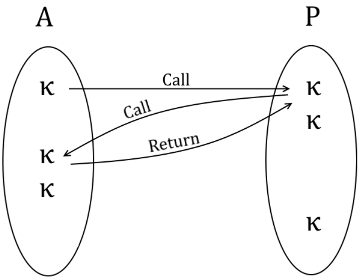
Raisonnement sur les attaques

Modèle programme-attaquant

- Il faut concevoir un **modèle** d'attaquant
- Une attaque est un jeu d'opposition entre un **programme partiel** et un **attaquant**

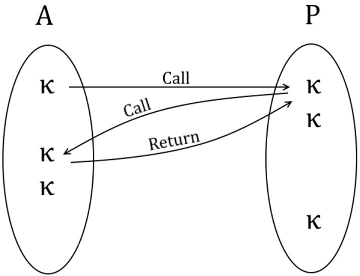
A	κ_0		κ_2	κ_3	
P		κ_1			κ_4

← Scénario d'attaque précis



Modèle programme-attaquant

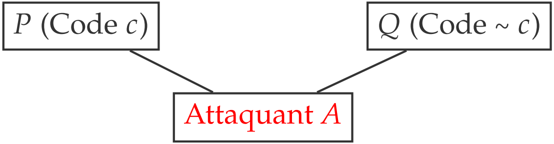
- **Déroulement** : celui qui possède le composant *main* commence. Actions internes illimitées, action externe à chaque tour.



Une séquence d'**actions** est une **trace**.

Modèle programme-attaquant

- **Objectif (Attaquant)** : distinguer le programme partiel P et l'une de ses variantes Q.



Jeu de **distinction**.

- **Fin** :
  - Le programme provoque sa terminaison
  - L'un des joueurs provoque une divergence

Théorème final : compilation compartimentée sécurisée

Définition

- Compilation Compartimentée Sécurisée
  - Préservation de l'abstraction dans notre contexte

Source (κ_0, κ_1) → Cible $(\kappa_0 \downarrow, \kappa_1 \downarrow)$ ↞ Attaquant $(\kappa_2 \downarrow, \kappa_3 \downarrow)$

```
procs 0 {
  0[0] := 1;
  Call 1.0(0);
  Exit;
}
procs 1 {
  Call 1.1(1);
  Exit;
}
```

```
Nop;
Mov raux1 raux2;
Jump raux1;
 Call 0 0;
Load raux2 raux3;
Mov raux2 raux3;
 Call 0 0;
Nop;
 Call 0 1;
 Halt;
```

Définition

- Obtenu par une autre propriété :
  - Abstraction complète structurée *(Structured Full Abstraction)*
    $\Rightarrow$ Attaque de bas niveau $\mapsto$ Attaque de haut niveau

Source (κ_0, κ_1) $\longrightarrow$ Cible $(\kappa_0 \downarrow, \kappa_1 \downarrow)$ $\twoheadleftarrow$ Attaquant $(\kappa_2 \downarrow, \kappa_3 \downarrow)$

Preuve (papier)

We can first apply trace decomposition (Lemma 4.5) to a and $P\!\downarrow$ to get a trace $t_i \in \mathit{Tr}_{\circ s}(P)$ that ends with $\checkmark$, such that $t_i \in \mathit{Tr}_{\bullet s}(a)$. Call t_p the longest prefix of t_i such that $t_p \in \mathit{Tr}_{\circ s}(Q\!\downarrow)$. Because trace sets are prefix-closed by construction, we know that $t_p \in \mathit{Tr}_{\circ s}(P\!\downarrow) \cap \mathit{Tr}_{\bullet s}(a)$.

Moreover, t_p is necessarily a *strict* prefix of t_i: otherwise, we could apply trace composition (Lemma 4.6) and get that $a[Q\!\downarrow]$ terminates, a contradiction. So there exists an external action $E\alpha$ such that trace "$t_p.E\alpha$" is a prefix of t_i. Now $E\alpha$ cannot be a context action, or else trace extensibility (Lemma 4.4) would imply that "$t_p.E\alpha$" is a trace of $\mathit{Tr}_{\circ s}(Q\!\downarrow)$, which is incompatible with t_p being the *longest* prefix of t_i in $\mathit{Tr}_{\circ s}(Q\!\downarrow)$. Therefore, $E\alpha$ is a program action, i.e., there exists γ_1 such that "$E\alpha = \gamma_1!$". Intuitively, $P\!\downarrow$ and $Q\!\downarrow$ take the same external actions until the end of t_p, where $P\!\downarrow$ takes external action "$\gamma_1!$" and $Q\!\downarrow$ does not (it takes either a different action $\gamma \neq \gamma_1$ or no external action at all).

Now, let t_c be the canonicalization of trace t_p, i.e., $t_c = \zeta_\circ(t_p)$. By canonicalization (Lemma 4.8), "$t_c.\gamma_1!$" $= \zeta_\circ(t_p.\gamma_1!)$ is a trace of $P\!\downarrow$. We can thus use apply definability (Assumption 4.9) to trace t_c and action γ_1, using $P\!\downarrow \in^\bullet s$ as a witness having trace "$t_c.\gamma_1!$". This yields a fully defined context $A \in^\circ s$ such that:

(1) $t_c \in \mathit{Tr}_{\bullet s}(A\!\downarrow)$,
(2) $\gamma_1 \neq \checkmark \Rightarrow (t_c.\gamma_1!.\checkmark?) \in \mathit{Tr}_{\bullet s}(A\!\downarrow)$,
(3) $\forall \gamma,\ \gamma'.\ \ (t_c.\gamma!.\gamma'?) \in \mathit{Tr}_{\bullet s}(A\!\downarrow) \Rightarrow \zeta(\gamma) = \zeta(\gamma_1)$.

$\gamma = \zeta(\gamma) \wedge \gamma_1 = \zeta(\gamma_1)$. Combined with (3), this entails that if $A\!\downarrow$ produced an action γ', we would have $\gamma = \gamma_1$, which is false. Hence, $A\!\downarrow$ doesn't produce any action: it goes into an infinite sequence of local transitions. We can again apply trace composition to get that $A\!\downarrow [Q\!\downarrow]$ diverges.

We finally apply separate compiler correctness (Corollary 4.3) to conclude the proof. □

5 Related Work

Fully abstract compilation Fully abstract compilation was introduced in the seminal work of Martín Abadi [1] and later investigated by the academic community. (Much before this, the concept of full abstraction was coined by Milner [46].) For instance, Ahmed *et al.* [9]–[11] proved the full abstraction of type-preserving compiler passes for functional languages and devised proof techniques for *typed* target languages. Abadi and Plotkin [6] and Jagadeesan *et al.* [33] expressed the protection provided by a mitigation technique called address space layout randomization as a probabilistic variant of full abstraction. Fournet *et al.* [29] devised a fully abstract compiler from a subset of ML to JavaScript.

Patrignani *et al.* [43], [55] were recently the first to study fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting hardware architectures featuring a new coarse-grained isolation mechanism. They also

Preuve (Coq)

```
(* We suppose that a[P↓] terminates and a[Q↓] diverges *)
Lemma structured_full_abstraction_aux_proof :
  forall a P Q s,
  let ap := (LL_context_application a COMPILE_PROG P↓) in
  let aq := (LL_context_application a COMPILE_PROG Q↓) in
  cprogram_terminates ap -> cprogram_diverges aq ->
  structured_full_abstraction_aux a P Q s.
Proof.
  unfold structured_full_abstraction_aux.
  intros a P Q s ap_terminates aq_diverges WF_s
    H_shP H_shQ H_PFD H_QFD.
  (* We consider a ∈• s distinguishing P ∈• s and Q ∈• s *)
  intros H_a. destruct H_a as [H_sha H_low_neq].
  inversion H_low_neq as [ap aq H_behavior ap_eq aq_eq].
  (* Goal : build a full-defined A ∈• s such that A[P] ≁ A[Q] *)
  (* We first apply trace decomposition *)
  assert (H_shq := H_shQ).
  apply shape_closed_under_compilation_program in H_shq.
  assert (H_shp := H_shP).
  apply shape_closed_under_compilation_program in H_shp.
  pose (trace_decomposition s (COMPILE_PROG P↓)
    H_shp a H_sha ap_terminates) as t_decomposition.
  destruct t_decomposition as [ti H_decomposition].
  (* We call tp the longest prefix of ti such that tp ∈ Tr•s(a) *)
  assert (exists tp, is_longest_prefix_of tp ti
    (COMPILE_PROG Q↓) s) as tp_exists.
  Case "Proof of existence of tp".
  { exists (longest_prefix_of ti (COMPILE_PROG Q↓) s).
    apply longest_prefix_of_spec. }
  destruct tp_exists as [tp H_tp].
  unfold is_longest_prefix_of in H_tp.
  destruct H_tp as [H_tp H_tp3].
  destruct H_tp as [H_tp H_tp2].
  unfold is_a_prefix_of in H_tp. inversion H_tp as [v H_prefix].
  assert (is_a_prefix_of tp ti) as H_tp1.
  { rewrite <- H_prefix. unfold is_a_prefix_of.
    exists v. reflexivity. }
  destruct H_decomposition as [t' [o]].
  destruct H as [H_tEnd H_tSets].
  destruct H_tSets as [H_tSets1 H_tSets2].
  (* We know that tp ∈ Tr•s(P↓) ∩ Tr•s(a) *)
  assert ((in_Traces_p tp (COMPILE_PROG P↓) s)
       /\ (in_Traces_a tp a s)) as H_tp_in_Pa.
  Case "Proof of tp ∈ Tr•s(P↓) ∩ Tr•s(a)".
```

```
    assert (cprogram_terminates
      (LL_context_application a COMPILE_PROG Q ↓)) as H_absurd.
    { apply t_compositionR. exists t'. exists o. apply H_tEnd. }
    assert (
      cprogram_terminates
        (LL_context_application a COMPILE_PROG Q ↓) /\
      cprogram_diverges
        (LL_context_application a COMPILE_PROG Q ↓))
      as absurd.
    split. apply H_absurd. apply aq_diverges.
    apply (LL_program_behavior_exclusion
      (LL_context_application a COMPILE_PROG Q ↓)) in absurd.
    contradiction. }
  (* There exists Ea such that tp.Ea such that tp.Ea
     is a prefix of ti *)
  assert (exists gl o, is_a_prefix_of (tp++[Ext gl o]) ti) as Ea_exists.
  { pose (strict_prefix_continuation tp ti s (COMPILE_PROG P↓)
      H_tp1 strict_prefix H_tSets1) as lemma.
    destruct lemma as [gl [o_origin [H_prefix']]].
    exists gl. exists o_origin. unfold is_a_prefix_of.
    exists H_prefix'. apply H. }
  destruct Ea_exists as [gl H_EaExists].
  destruct H_EaExists as [origin_Ea H_EaExists].
  remember [Ext gl origin_Ea] as Ea.
  rename HeqEa into H_gl.
  (* Ea is a program action *)
  assert (origin_Ea = ProgramOrigin)
    as H_program_action.
  { destruct origin_Ea.
    (* Ea cannot be a context action *)
    pose (trace_extensibility tp s gl (COMPILE_PROG Q↓)
    H_shq a H_sha) as t_extensibility.
    destruct t_extensibility as [t_ext1 t_ext2].
    assert (in_Traces_p tp COMPILE_PROG Q ↓ s /\
      in_Traces_a (tp ++ [Ext gl ContextOrigin]) a s)
      as H_assert.
    { split. assumption. rewrite <- H_gl.
      apply (trace_sets_closed_under_prefix_context
        (tp++Ea) ti a s H_sha H_EaExists H_tSets2). }
    apply t_ext1 in H_assert.
    pose (H_tp3' := H_tp3).
    specialize (H_tp3' (tp ++ [Ext gl ContextOrigin])).
    rewrite H_gl in H_EaExists.
    specialize (H_tp3' (conj H_EaExists H_assert)).
    unfold is_a_prefix_of in H_tp3'. destruct H_tp3'.
```

- Preuve informelle ↦ Preuve formelle

Conclusion

Difficultés et problèmes

- Comprendre le projet
  - Entrer dans le sujet
  - Problématique abstraite
- Problèmes de développement
  - Manque d'anticipation
  - Adapter les concepts formels

Bilan

Ce que ce stage m'a apporté

- Découverte de l'environnement de la recherche (Séminaires, soutenance de thèse, conférence)
- Élargissement des connaissance théoriques et techniques

Ce que j'ai apporté

- Détection indirecte d'erreurs
- Garanties supplémentaires pour le projet

Démonstration : preuve avec Coq

- Un peu de choses concrètes :
  $\forall p, \neg(p \text{ termine}) \Rightarrow (p \text{ diverge})$

- Axiome du tiers exclu :
  $\forall P, P \vee \neg P$

Bibliographie

- B. C. Pierce, *Software Foundations*. University of Pennsylvania, Version 3.2, 2015.
- B. C. Pierce, *Types and programming languages*. Cambridge, Massachusetts, The MIT Press, 2002.
- Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. C. Pierce, B. Eng *Beyond Good and Evil : Formalizing the Security Guarantees of Low-Level Compartmentalization*. 2016.
- A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach., *Micro-policies : Formally verified, tag-based security monitors*, In 36th IEEE Symposium on Security and Privacy (Oakland S&P), 2015.

Bibliographie

- X. Rival., *Operational Semantics - Semantics and applications to verification (Lecture slides)*, École Normale Supérieure, 2015. (Slides : `http://www.di.ens.fr/~rival/semverif-2015/sem-02-trace.pdf`)
- A. M. Pitts ., *Semantics of Programming Languages (Lecture notes)*, University of Cambridge, 2002. (Notes : `http://www.inf.ed.ac.uk/teaching/courses/lsi/sempl.pdf`)
- M. Abadi., *Protection in programming-language translations.*, Research Report 154, SRC, 2015.
- Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, Steve Zdancewic, *SoftBound : Highly Compatible and Complete Spatial Memory Safety for C* , University of Pennsylvania.

Bibliographie

Catalin Hritcu., *Micro-Policies : Formally Verified, Tag-Based Security Monitors (Talk Rennes)*, Inria Paris - Prosecco, 2015. (Slides : `http://prosecco.gforge.inria.fr/personal/hritcu/talks/Micro-Policies-Rennes.pdf`)

Sitographie

- https://fr.wikipedia.org/wiki/Plan_Calcul
- https://fr.wikipedia.org/wiki/Institut_national_de_recherche_en_informatique_et_en_automatique
- http://www.inria.fr/centre/paris/presentation/une-forte-reconnaissance
- https://fr.wikipedia.org/wiki/IRILLS
- http://prosecco.gforge.inria.fr/people.php
- http://www.inria.fr/institut/inria-en-bref/chiffres-cles

Sitographie

- http://www.inria.fr/institut/inria-en-bref/chiffres-cles
- http://www.inria.fr/institut/strategie
- http://www.inria.fr/institut/partenariats/partenariats-industriels
- https://en.wikipedia.org/wiki/Coq
- http://www.inria.fr/centre/paris/recherche