Sécurité du logiciel, troisième cours

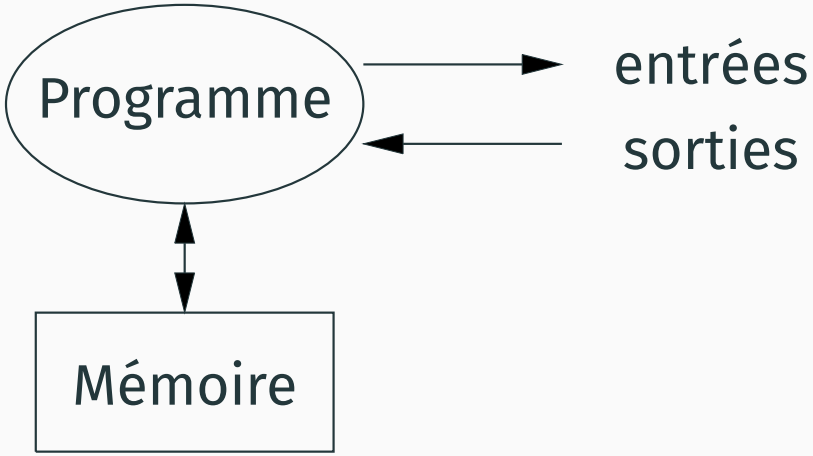
Isolation logicielle

Xavier Leroy

2022-03-24

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

Dans un monde simple…



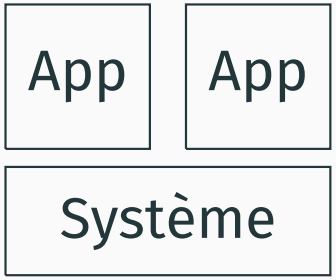
L'exécution se produit comme décrit par le code du programme.

Le code est intègre (pas modifié de l'extérieur).

Les données en mémoire sont intègre (telles qu'écrites par le programme) et confidentielles (pas de fuites vers l'extérieur).

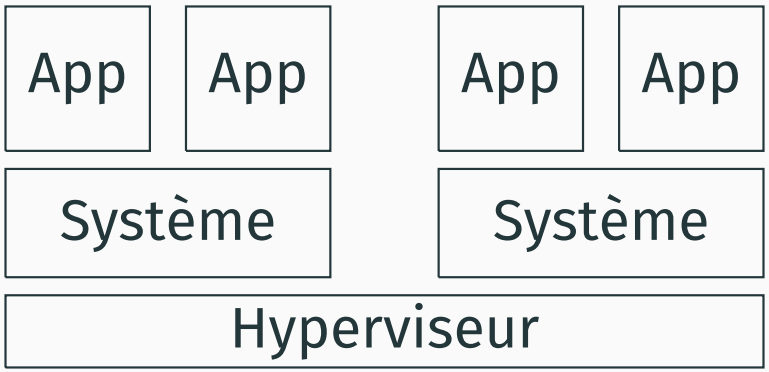
Les interactions avec le monde extérieur se font uniquement par des opérations d'entrée/sortie bien identifiées.

En réalité…



Temps partagé, multiprocesseurs : plusieurs processus partagent la même mémoire.

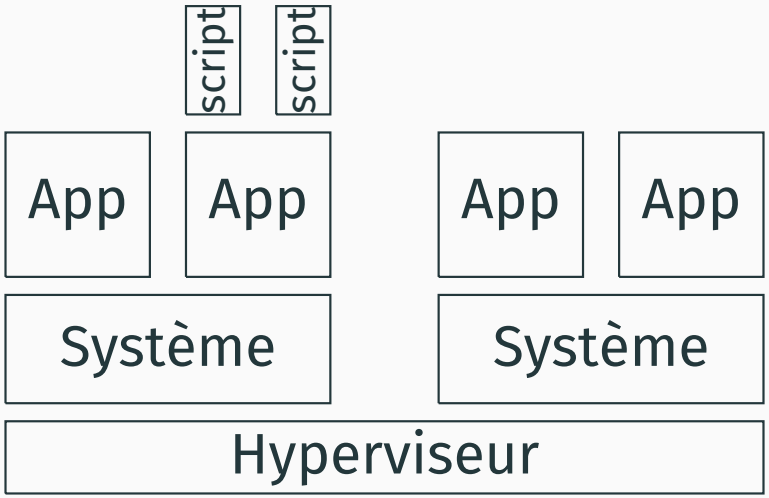
En réalité…



Temps partagé, multiprocesseurs : plusieurs processus partagent la même mémoire.

Virtualisation : plusieurs systèmes d'exploitation partagent la mémoire et les périphériques.

En réalité…

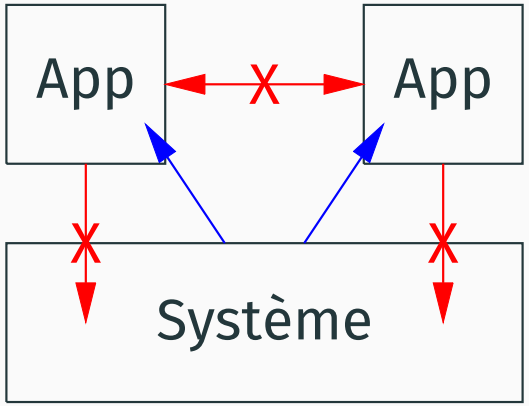


Temps partagé, multiprocesseurs : plusieurs processus partagent la même mémoire.

Virtualisation : plusieurs systèmes d'exploitation partagent la mémoire et les périphériques.

Scripts, macros, extensions, *plug-ins*, *applets* : plusieurs codes d'origine différentes s'exécutent au sein du même processus (p.ex. le navigateur Web).

Le problème de l'isolation



Comment garantir qu'un programme s'exécute sans être perturbé par les programmes ≪au même niveau≫ ou ≪au dessus≫ ?

Sans pertubation ≈ intégrité du code et son exécution ;
intégrité et confidentialité des données.

Sandboxing vs. *Shielded execution*

Sandboxing : ≪le bac à sable≫

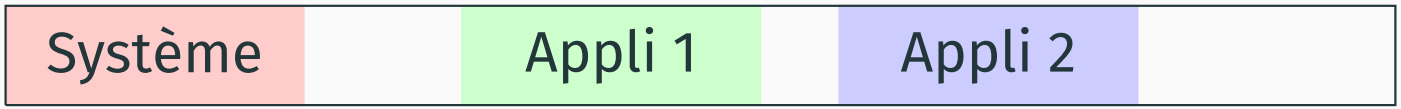
Un code d'origine douteuse est isolé pour l'empêcher de perturber les autres.

Shielded execution : ≪la chambre forte≫

Un code critique (système d'exploitation, cryptographie, …) est isolé pour éviter qu'il ne soit perturbé par les autres.

Isolation par le matériel

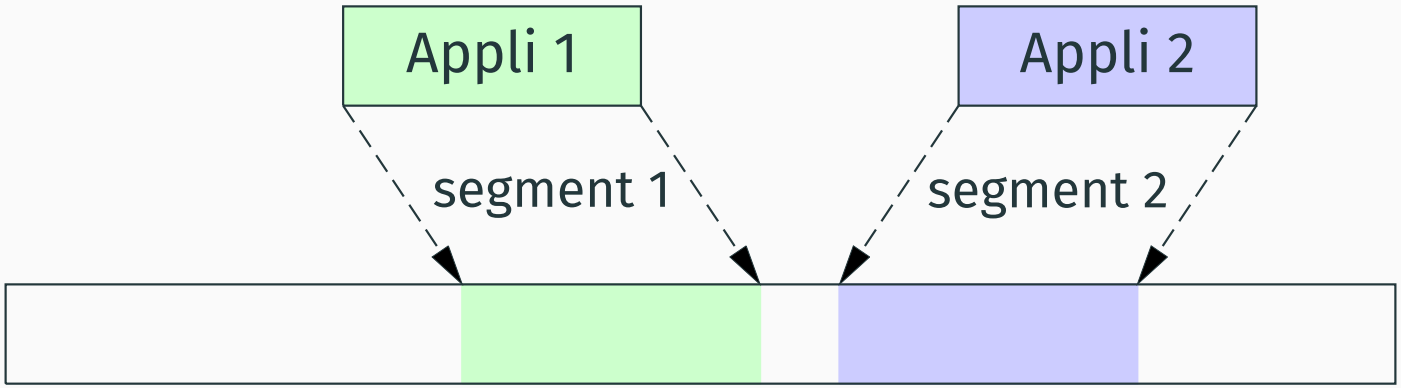
Une mémoire partagée entre plusieurs programmes



Temps partagé et multiprocesseurs : plusieurs programmes et le système d'exploitation partagent la même mémoire vive.

Comment empêcher un programme de lire ou écrire dans les données ou le code d'un autre programme ? ou du système ?

La segmentation (Burroughs B5000, 1961)



Un segment = une paire (adresse de base, taille).

Traduction adresse logique (programme) $\rightarrow$ adresse physique :

si $0 \leq$ adresse logique $<$ taille

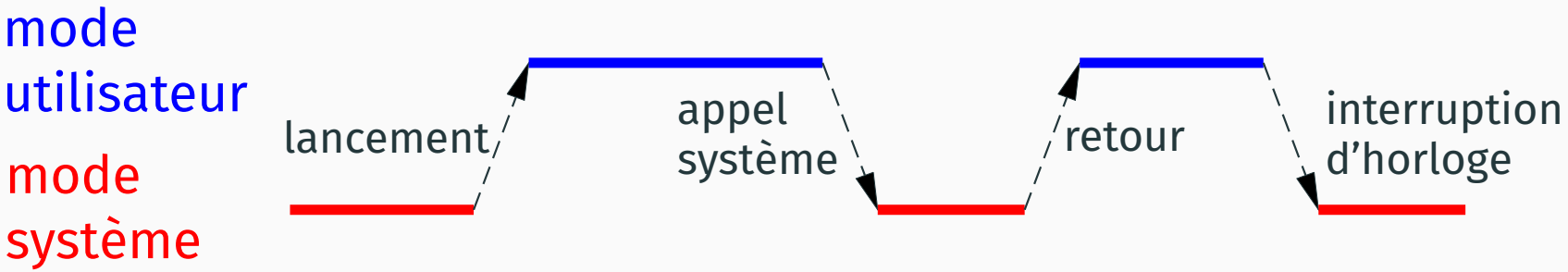
alors adresse de base $+$ adresse logique

sinon erreur de segmentation

Un segment pour chaque processus, 2 à 2 disjoints

(+ un segment égal à toute la mémoire pour le système).

Mode système, mode utilisateur

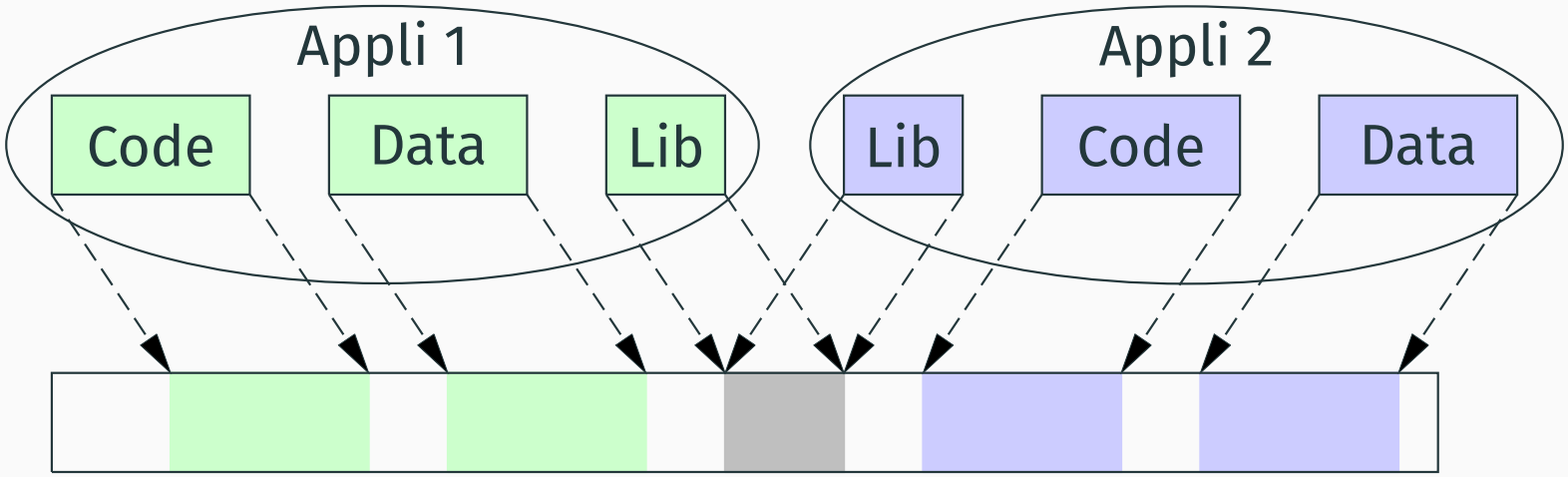


Le processeur fonctionne dans deux modes : système et utilisateur.

Les instructions privilégiées, dont la modification des registres de segments, s'exécutent uniquement en mode système.

Une instruction spéciale permet aux processus utilisateur de passer en mode système en sautant à un point d'entrée fixé dans le système.

Plusieurs segments par programme (Multics, 1969)

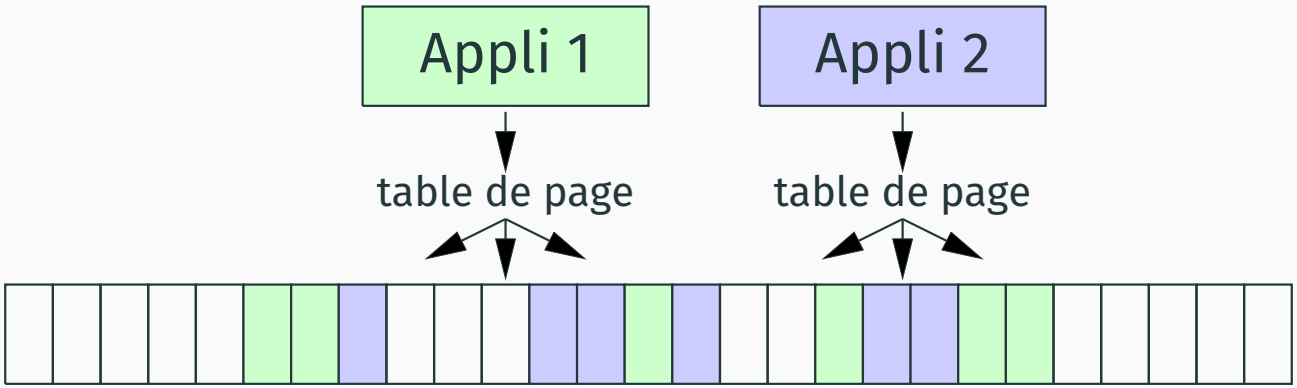


Chaque processus peut accéder à plusieurs segments : code, données, pile, ...

Un segment de code en lecture seule peut être partagé entre plusieurs processus $\rightarrow$ bibliothèques partagées.

Un segment de données peut être partagé entre plusieurs processus $\rightarrow$ communication inter-processus.

La pagination (IBM S/360-67, 1966)



La mémoire est divisée en pages (typiquement : 4 kilo-octets).

Pour chaque processus, une table de pages fait correspondre les pages de mémoire virtuelle en pages de mémoire réelle.

Les pages d'un processus ne sont pas nécessairement contiguës en mémoire réelle.

La pagination

Chaque page a des droits d'accès : lecture, écriture, exécution.

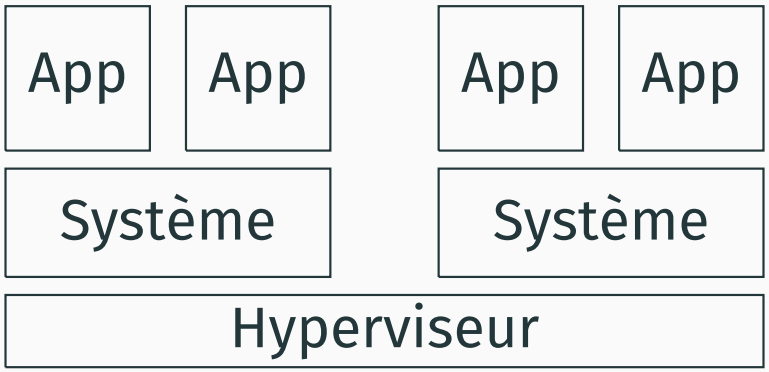
Lecture à la demande des pages de code depuis un fichier.

Écriture sur le disque (*swapping*) et expulsion de pages de données quand on manque de mémoire réelle.

Les instructions qui manipulent la table de page sont privilégiées.

Gestion complexe des tables de page, impliquant circuits, microcode, et logiciel système.

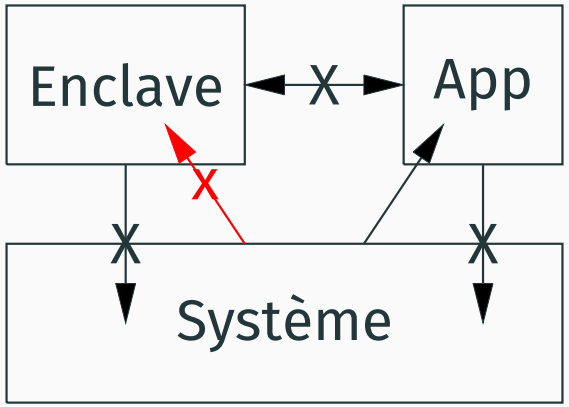
La virtualisation (IBM S/370, 1970 ; *cloud computing*, 2002)



Exécuter un système d'exploitation non pas sur une vraie machine, mais sur une machine virtuelle émulée par un logiciel hyperviseur :

- Le système d'exploitation tourne en mode utilisateur.
- Ses instructions privilégiées sont interceptées par l'hyperviseur.
- L'hyperviseur émule les composants d'une vraie machine (tables de pages, périphériques, bus, …).

Les enclaves (Arm TrustZone, 2004 ; Intel SGX, 2015)



Un processus lancé par / communiquant via un système, mais protégé contre le système par des mécanismes cryptographiques matériels :

- Pages mémoires chiffrées
- Points d'entrée bien délimités
- Accepte seulement des programmes signés
- Mécanisme d'attestation cryptographique.

Bilan sur l'isolation par le matériel et le logiciel système

En positif :

- Garantit la séparation des espaces mémoire.
- Les applications ne peuvent pas contourner le contrôle d'accès implémenté par le système.
- Permet de nombreuses formes de virtualisation et de contrôle des ressources.

En négatif :

- Coûts élevés des communications entre applications, du lancement d'un processus ou d'une machine virtuelle.
- Peu flexible.

Y-a-t'il des mécanismes plus légers et plus souples ?
notamment pour isoler les scripts au sein d'une application ?

Isolation logicielle des fautes

Principe : moniteurs de référence intégrés (*Inline Reference Monitors*)

Le moniteur de référence =

le logiciel qui met en œuvre le contrôle d'accès.

Deux implémentations typiques :

1. Moniteur séparé et isolé,
   typiquement dans le noyau du système.
2. Moniteur intégré (*inlined*) dans le code de l'application,
   typiquement pendant la compilation, ou par réécriture
   du code machine.

Application : segmentation des accès aux données

On simule une architecture segmentée en transformant le code de chaque lecture ou écriture de donnée.

En pseudo-code :

```
x = *p;    →    if (p >= seg.size) abort();
                x = *(seg.base + p);
```

En code machine :

```
                    cmp r1, rsize
load r0, [r1]   →   jae abort
                    add rtmp, rbase, r1
                    load r0, [rtmp]
```

(`rbase`, `rsize`, `rtmp` sont des registres réservés à cet usage)

Remplacement du test de bornes par un masquage

En cas d'accès hors des bornes du segment, au lieu de signaler une erreur, on accède à une adresse quelconque dans le segment.

P.ex. on peut accéder à l'adresse modulo la taille du segment :

```
x = *p;    →    x = *(seg.base + p % seg.size);
```

Si la taille du segment est une puissance de 2, cela revient à masquer avec `seg.mask` = `seg.size` − 1 :

```
x = *p;    →    x = *(seg.base + p & seg.mask);
```

```
                and rtmp, r1, rmask
load r0, [r1]  →  add rtmp, rbase, rtmp
                load r0, [rtmp]
```

Quand effectuer la transformation des accès mémoire ?

Pendant la compilation depuis un langage source ou la génération de code machine depuis une représentation intermédiaire (LLVM, etc).

Exemple : WebAssembly et son espace d'adressage de 4 Go.

Difficulté : la compilation fait partie de la base de confiance (TCB).

Par réécriture d'un code machine déjà compilé.

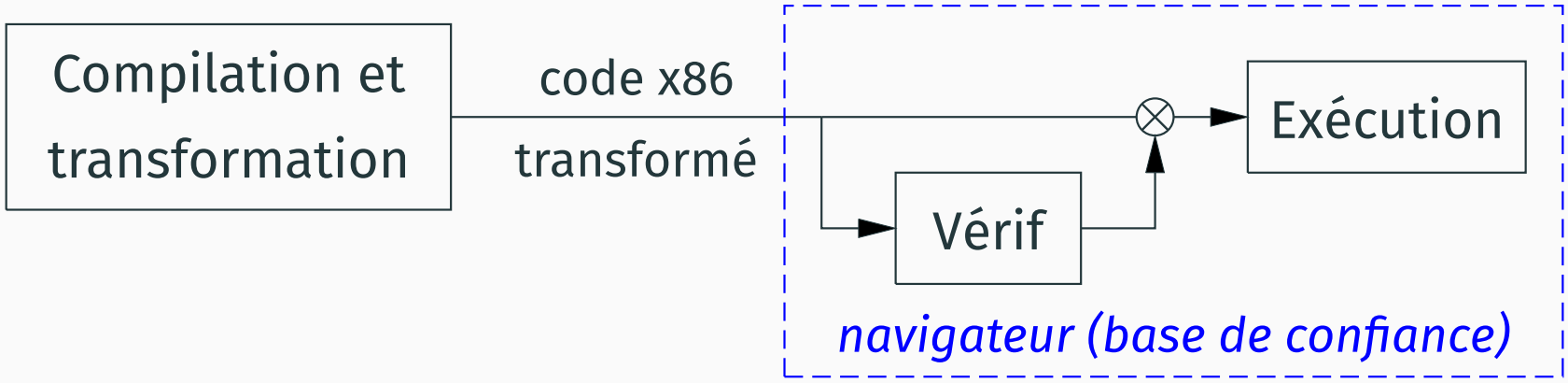
Difficultés : désassemblage et réassemblage ;
trouver suffisamment de registres libres.

Vérifier *a posteriori* la transformation

Vérifier sur le code machine que la transformation a été appliquée (tous les accès mémoire sont protégés) :

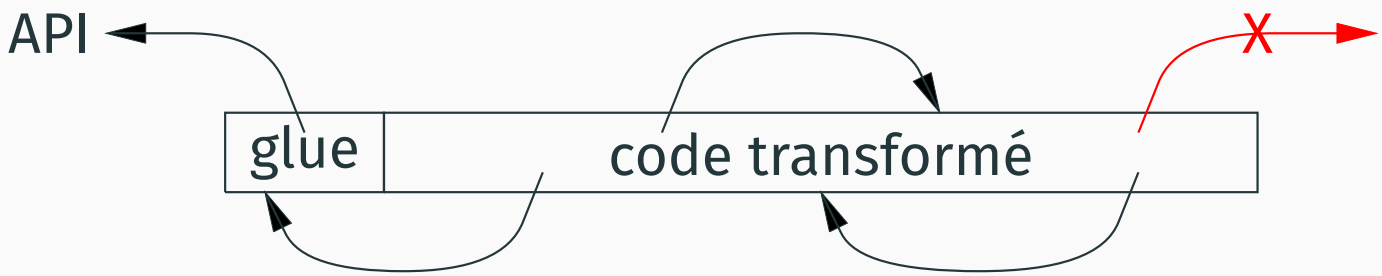
- désassemblage, analyse du code
- mais pas de réécriture et de réassemblage
- et pas besoin de trouver des registres libres.

Exemple : le système Native Client (NaCl).



(Yee et al, *Native Client : A sandbox for portable, untrusted x86 native code*, 2009).

Vérification des branchements



Garantir que l'exécution reste à l'intérieur du code transformé.

(Exception : on peut appeler des fonctions de l'application hôte en passant par du code «glue», mis au début du segment de code.)

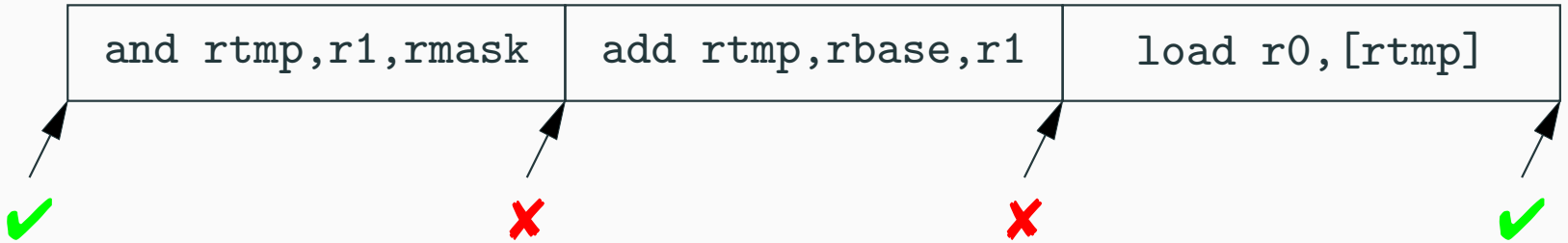
Branchements avec offsets constants : vérification statique.

Branchements calculés, retours de fonctions : segmentation.

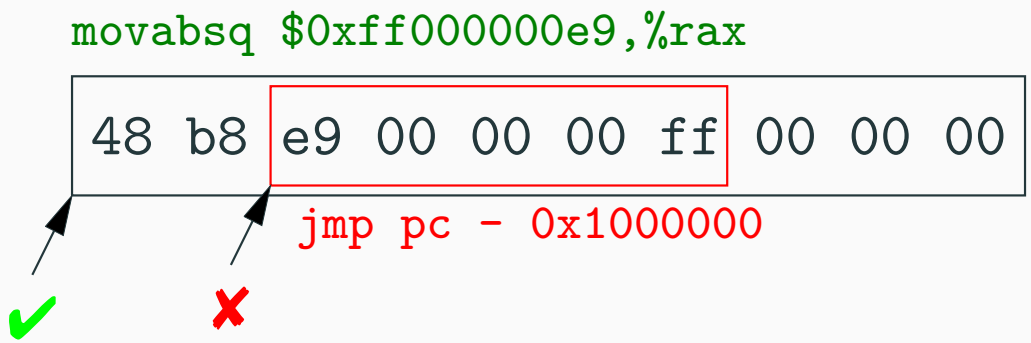
```
                          and rtmp, r0, rcodemask
jump r0          →        add rtmp, rcodebase, rtmp
                          jump r0
```

Vérification des cibles de branchements

La cible d'un branchement ne doit pas être au milieu d'une séquence instrumentée :

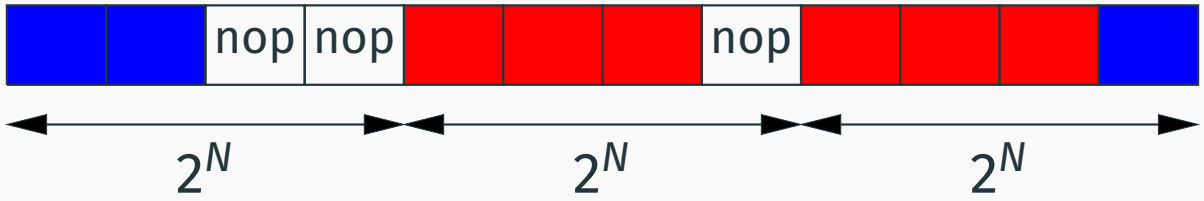


La cible d'un branchement ne doit pas être au milieu d'une instruction du processeur :



Vérification des cibles de branchements

Vérifier que les instructions (en bleu) et les blocs d'instructions produits par transformation (en rouge) sont sont entièrement contenus dans des blocs de 2^N octets.



(Toujours possible par ajout de `nop` en produisant le code.)

Mettre les N bits de poids faible de `rcodemask` à 0 de sorte que tous les branchements calculés se font à des multiples de 2^N.

```
                          and rtmp, r0, rcodemask
jump r0          →        add rtmp, rcodebase, rtmp
                          jump r0
```

Bilan sur l'isolation logicielle des fautes

Un exemple intéressant d'isolation par transformation de code, sans utiliser de mécanismes *hardware*.

Utilisation principale : *sandboxing* de bibliothèques complexes dans des applications sensibles (navigateur Firefox).

Performances acceptables : $\approx$ +20% en temps d'exécution
($\approx$ +10% si les lectures mémoire ne sont pas protégées).

Sécurité pas facile à garantir
$\rightarrow$ preuve formelle du vérificateur (Morrisset et al, *Rocksalt*, 2012).

Isolation par le langage

Isolation par le langage de programmation

Une approche qui se développe dans les années 1990 pour répondre aux besoins du code mobile :

- des bouts de programmes téléchargés sur Internet,
- en lesquels on n'a pas confiance,
- exécutés automatiquement dans un navigateur, un lecteur de mail, un traitement de texte, etc.

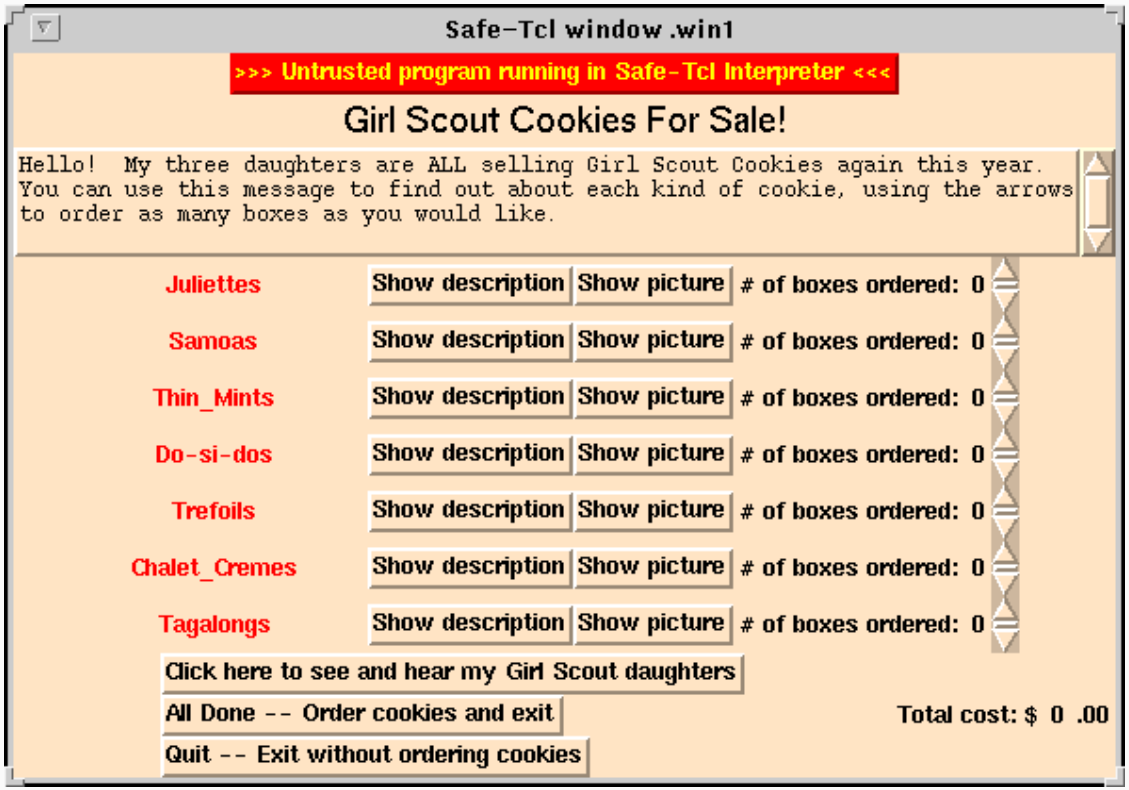
Les composantes de l'approche :

- un langage de programmation et un modèle d'exécution sûrs (typage fort, dynamique ou statique);
- des interfaces logicielles (API) restreintes, limitant les interactions entre le code mobile et l'environnement d'exécution.

Un précurseur : *Email with a mind of its own*

(N. Borenstein, *EMail With A Mind of Its Own : The Safe-Tcl Language for Enabled Mail*, 1991, 1997)

Idée : pouvoir mettre en pièce jointe d'un mail un programme de type «script» : formulaire interactif, carte de vœux animée, etc.



Le langage et l'API Safe-TCL

Le langage de script TCL avec une API modifiée :

- Interaction via l'interface graphique TK ou le terminal.
- Pas d'accès aux fichiers ou aux commandes système.
- Exception : possibilité d'envoyer du mail ou d'imprimer un document, après accord de l'utilisateur.

Les *applets* Java (1995)

Programmes Java compilés en code JVM, téléchargés sur le Web et exécutés dans le navigateur.

Les *applets* Java (1995)

Exécution efficace par vérification du bytecode JVM
($\approx$ typage statique) puis interprétation avec peu de vérifications dynamiques.

Même API pour les applets que pour les applications locales.
(*Write once, run anywhere.*)

Contrôle d'accès intégré à l'API :
les permissions dépendent de l'origine du code (local / applet) et des signatures cryptographiques qu'il porte.

JavaScript (1995)

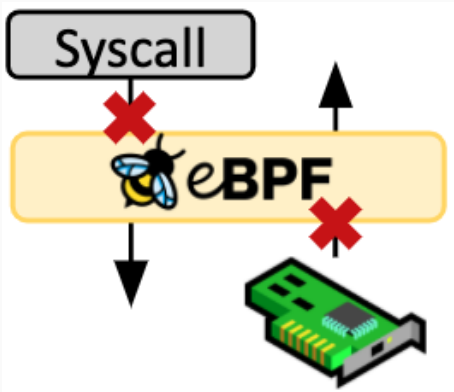
Initialement : un langage de script de plus, typé dynamiquement, interprété, pour rendre les pages Web plus interactives.

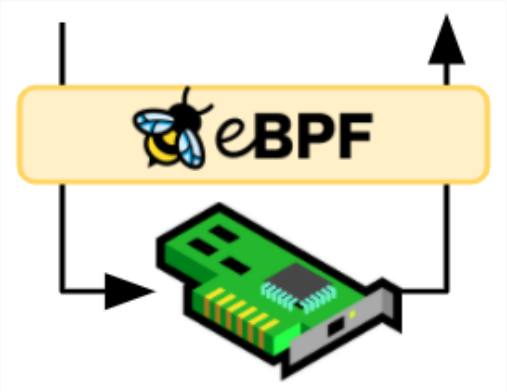
Depuis 2004 : le langage d'implémentation des applications ≪AJAX≫ s'exécutant dans le navigateur.

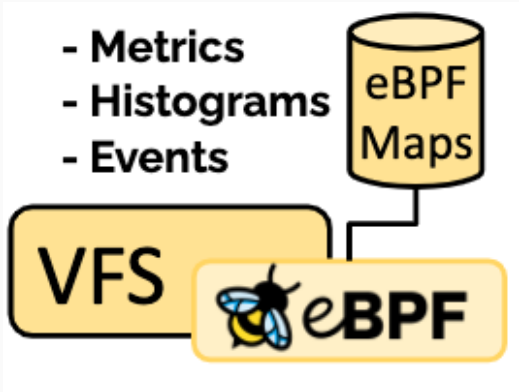
Aujourd'hui : un des langages les plus utilisés et les mieux compilés.

Modèle de sécurité : API limitées + politique de la même origine.

Berkeley Packet Filter (BPF, 1992 ; eBPF, 2014)







Du code pour une machine virtuelle à registres qui peut être injecté dans les noyaux BSD ou Linux, à des fins de filtrage réseau, monitoring, etc.

Accès très contrôlé aux structures de données du noyau.

Sûreté de l'exécution garantie par analyse statique du code : analyse de valeurs, sûreté des accès mémoire, terminaison.

Sûreté de l'exécution

Deux grands modèles :

1. exécution défensive (tests dynamiques) [JavaScript, TCL]
2. vérification statique (avant l'exécution) [Java, BPF]
   + exécution ≪offensive≫ (moins de tests dynamiques)

Indépendant de interprété / compilé.

Indépendant du format du code mobile :
code source ou code compilé pour une machine virtuelle.

Sécurisation de l'API

Limiter les fonctionnalités visibles par le code mobile :

- Exposer uniquement les fonctions non dangereuses.
- Utiliser des modificateurs de visibilité
  (en Java : `private`, `protected`, package-local).
- Utiliser l'abstraction de types.

($\rightarrow$ cours du 7 avril)

Intégrer du contrôle d'accès aux fonctions de l'API.

- Exemple : le `SecurityManager` en Java.

Sécurité de l’API Java

Le modèle de capacités de Java

À chaque méthode est associé un ensemble de capacités, aussi appelées permissions :

Fichiers : lire, écrire, exécuter, effacer (en fonction du nom du fichier)

Réseau : ouvrir une connexion sortante, accepter les connexions entrantes (en fonction du nom de l'autre machine et du port)

Runtime : arrêter le programme, charger du code natif, définir un *class loader*, définir un *security manager*, …

GUI : accéder au presse-papier, à la file d'événements, …

(et bien d'autres encore).

Associer des capacités à chaque méthode

Les capacités sont déterminées par le *class loader* qui a chargé le code de la méthode. Typiquement :

- Code chargé depuis des fichiers locaux : tous les droits.
- Code chargé depuis le Web : pas d'accès aux fichiers ni au runtime ; connexions réseau uniquement avec la machine d'où provient le code.
- Des capacités supplémentaires peuvent être accordées aux codes qui portent une signature cryptographique reconnue.

Contrôler les accès

Avant d'effectuer une opération dangeureuse, les méthodes de l'API appellent la méthode `checkPermission` (ou ses variantes `checkFile`, `checkAccept`, etc) de la classe `SecurityManager`.

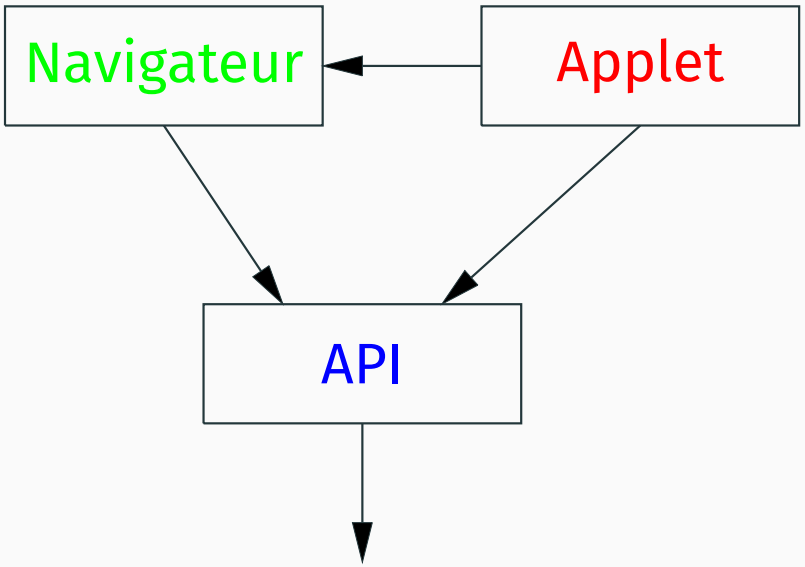
Si toutes les méthodes appelantes ont la permission requise, `checkPermission` revient normalement et l'accès est accordé.

Si l'une des méthodes appelantes n'a pas la permission requise, `checkPermission` lève une exception ; l'accès est refusé.

L'inspection de la pile d'appels

Cette vérification des permissions s'appelle l'inspection de la pile d'appels (*stack inspection*).

Elle permet à une API de se comporter différemment suivant qu'elle agit au nom d'un code digne de confiance (p.ex. le navigateur) ou d'un code douteux (une applet).



Navigateur $\rightarrow$ API :
accès accordé.

Applet $\rightarrow$ API :
accès refusé.

Applet $\rightarrow$ Navigateur $\rightarrow$ API :
accès refusé.

Amplification des privilèges

Une API peut avoir besoin d'effectuer une opération risquée quel que soit le contexte d'appel.

Exemple : pour afficher du texte dans une interface graphique, on peut avoir besoin de charger une police de caractères depuis un fichier local, même si on est appelé depuis une applet.

Amplification des privilèges

Le code de chargement de polices peut utiliser `doPrivileged` pour s'exécuter avec ses permissions à lui, même s'il est appelé depuis du code ayant moins de permissions.

```
void loadFontFile(String name) {
    // validation du paramètre
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            // ouvrir et lire le fichier de la fonte
            return null;
        }
    });
}
```

L'algorithme de contrôle d'accès

Pour chaque appel en cours présent dans la pile d'appels,
du plus récent au plus ancien :

> Soit *M* la méthode en cours d'exécution dans cet appel.
>
> *M* a-t'elle la permission requise ?
> Si non, lever l'exception `SecurityException`
>
> *M* a-t'elle amplifié ses permissions ?
> Si oui, l'accès est accordé

Le *thread* courant a-t'il démarré avec la permission requise ?
Si non, lever l'exception `SecurityException`
Si oui, l'accès est accordé.

Manipuler l'autorité (*confused deputy*)

L'inspection de pile protège l'appelé de l'appelant, mais parfois l'appelant (le *confused deputy*) fait trop confiance à l'appelé…

Exemple : (Abadi & Fournet, 2003)

```
class Trusted {
  // toutes permissions
  static void main() {
    String s =
      Applet.tempfile();
    File.delete(s);
  }
}
```

```
class Applet {
  // aucune permission
  static String tempfile() {
    return "/etc/passwd";
  }
}
```

Manipuler l'autorité (*confused deputy*)

Un autre exemple, impliquant le sous-classage :

(Abadi & Fournet, 2003)

```
class Trusted {
  // toutes permissions
  String tempfile = "/tmp/file";
  abstract void proceed();
  static void main() {
    File.create(tempfile);
    try {
      proceed();
    } finally {
      File.delete(tempfile);
    }
  }
}
```

```
class Applet extends Trusted {
  // aucune permission
  void proceed() {
    tempfile = "/etc/passwd";
  }
}
```

De la pile d'appels à l'historique de l'exécution

(M. Abadi & C. Fournet, *Access Control based on Execution History*, 2003.)

Idée : au lieu de contrôler les accès sur la base des méthodes en cours d'appel, les contrôler sur la base de toutes les méthodes qui ont été appelées jusqu'ici.

La machine a un état C contenant les permissions courantes.

Appel de méthode $\quad obj.m(\ldots)$

$$C := !C \cap \texttt{permissions}(obj.m)$$

Amplification de privilèges $\quad \texttt{grant}(P)\ \{\ B\ \}$

$$\texttt{let}\ \ C_0 = !C\ \ \texttt{in}\ \ C := !C \cup P;\ B;\ C := !C \cap C_0$$

Politiques de contrôle d'accès utilisant l'historique de l'exécution

Connaître l'historique de l'exécution permet d'implémenter des politiques plus fine, comme

> *Une applet peut lire un fichier local ou établir une connexion réseau, mais pas les deux.*

(L'ouverture d'un fichier enlève la permission ≪réseau≫, et l'établissement d'une connexion enlève la permission ≪fichiers≫.)

Bilan sur l'API Java

Une idée intéressante : le contrôle d'accès en fonction du contexte d'appel.

Un mécanisme imparfait : l'inspection de pile.

Des alternatives à l'inspection de pile, pas adoptées.

Pas d'API pour faire interagir l'applet et le document Web…

Sécurité de l’API JavaScript

Le modèle de sécurité de JavaScript

Langage typé dynamiquement + API restreintes, orientées Web :

- DOM : interface avec le document (la page Web affichée)
- XmlHttpRequest : connexion directe à des serveurs HTTP
- Cookies, Web Storage : stockage limité de données
- etc.

Protège assez bien le navigateur et le système d'exploitation des pages Web et des scripts malveillants.

Peu d'isolation entre scripts et entre scripts et pages Web.

Politique de la même origine (SOP, *same-origin policy*)

L'origine d'une page ou d'une `iframe` =
les parties (protocole, hôte, port) de son URL.

Un script ou un élément du DOM ont comme origine celle de la page qui les contient.

Politique de la même origine : un script peut seulement

- lire ou modifier un élément DOM ayant la même origine ;
- faire une connexion XmlHttpRequest sur le serveur d'origine (d'autres serveurs peuvent être autorisés via le protocole CORS) ;
- accéder aux cookies ayant même domaine et même chemin.

SOP n'empêche pas CSRF (*Cross-Site Request Forgery*)

Les liens `<a>`, les formulaires, les éléments inclus `<img>`, `<script>` ne sont pas soumis à la politique de la même origine.

Un script malveillant peut insérer dans le document un chargement d'image à une adresse quelconque :

```
<img width="0" height="0"
     src="https://vulnerable.service.com/email/change
         ?newmail=attacker@evil.net">
```

Si le document provient de `vulnerable.service.com` et si l'utilisateur a une session active, les cookies authentifiant la session sont transmis et l'adresse email est changée.

Quelques autres vulnérabilités de SOP

Les hôtes sont identifiés par leurs noms et non par leurs adresses IP $\rightarrow$ possibilités d'attaque via le service DNS.

Un script peut changer la partie ≪hôte≫ de l'origine de la page (vers un sur-domaine seulement).

```
                                    // Avant: test.example.com
document.domain = "example.com";
                                    // Après: example.com
```

L'accès au site de test donne accès au site de production…

JavaScript : un langage très dynamique

Une fonction peut être redéfinie après coup :

```
f = function(arg) { /* new body */ };
```

Les méthodes d'un objet peuvent être modifiées après création :

```
obj.meth = function(arg) { /* new body */ };
```

Le prototype d'un objet et les méthodes du prototype aussi :

```
obj.prototype = new MyClass();
Object.prototype.__defineSetter__('x', myfunc);
```

Cela permet des attaques d'un script sur un autre, mais aussi l'ajout de protections sur l'API ou sur un script…

Sécuriser davantage les fonctions de l'API

(Phung, Sands, Chudnov, *Lightweight Self-Protecting JavaScript*, 2009).

Idée : charger en premier dans la page un script qui redéfinit des fonctions de l'API pour leur ajouter des contrôles d'accès.

```
<html>
  <head> <script src="./policy.js"></script> </head>
  <body>
    <!-- contenu de la page, inchangé -->
  </body>
</html>
```

Redéfinir des fonctions de l'API

Exemple : empêcher `windows.open` d'ouvrir plus de 5 fenêtres.

```
function () {
    var orig = windows.open;
    var num_windows = 0;
    var wrapped = function () {
        if (++num_windows >= 5) abort();
        orig();
    }
    windows.open = wrapped;
} ();
```

Note : `orig` et `num_windows` ont pour portée le bloc de la fonction anonyme, et ne sont pas visibles ailleurs dans la page.

Redéfinition en style programmation par aspects

```
var wrap = function(pointcut, Policy) {
  // Override the prototype of the object if available
  var source = (typeof(pointcut.target.prototype) != undefined)
               ? pointcut.target.prototype : pointcut.target;
  var method = pointcut.method;
  // Save reference to the original method
  var original = source[method];
  // Weave the policy with the original method
  var aspect = function() {
    var invocation = {object:this, args:arguments};
    return Policy.apply(invocation.object,
        [{arguments:invocation.args, method:method,
          proceed:function(){
            return original.apply(invocation.object, invocation.args);}}]);
  }
  // Redefine the method
  source[method] = aspect;
  return aspect;
}
```

Vulnérabilités dans la redéfinition

(Magazinius, Phung, Sands, *Safe wrappers and sane policies for self protecting JavaScript*, 2012.)

```
return Policy.apply(invocation.object,
    [{arguments:invocation.args, method:method,
      proceed:function(){
        return original.apply(invocation.object, invocation.args);}}]);
```

L'attaquant peut récupérer la méthode non sécurisée `original`, p.ex. en redéfinissant `Function.apply` ou le *setter* pour `proceed`.

```
var recover_builtin;
Object.prototype.__defineSetter__('proceed',
            function(o) { recover_builtin = o });
```

(Voir Magazinius et al pour des contre-mesures possibles.)

Protéger un script du reste de la page

(K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, *Defensive JavaScript : building and verifying secure Web components*, 2014.)

```
<html><body>
   <script src="attack1.js"></script>
   <script src="sensitive.js"></script>
   <script src="attack2.js"></script>
</body></html>
```

Contexte : un script implémente une opération sensible (chiffrement de bout en bout, signature, communication avec un gestionnaire de mots de passe, etc.) Il peut se retrouver avec des script malveillants dans une même page Web.

Exemple : messages authentifiés

```
var f = function(msg) {
    var key = "...";
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://logging.example.com", false);
    xhr.send(Crypto.HMAC(key, msg) + "," + msg);
}
```

La clé `key` utilisée pour signer le message est incluse dans le script.

Elle ne peut pas être lue par les autres scripts de la page car sa portée est locale à la fonction `api`.

Mais `f.toSource()` renvoie le texte de la fonction (type `string`), d'où ils peuvent extraire la clé ! ($\rightarrow$ séminaire S. Blazy)

Cacher le source de la fonction

```
var f = (function () {
    var g = function(msg) {
        var key = "...";
        // envoi du message
    }
    return function(msg){ return g(msg); }
}) ();
```

Maintenant, `f.toSource()` révèle uniquement le *wrapper* `function(msg){return g(msg);}` mais pas la fonction `g`.

Appels de fonctions externes

```
...
var key = "...";
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://logging.example.com", false);
xhr.send(Crypto.HMAC(key, msg) + "," + msg);
...
```

Le script attaquant peut redéfinir `Crypto.HMAC` pour faire ≪fuiter≫ la clé :

```
var hmac = Crypto.HMAC; var leaked_key;
Crypto.HMAC =
  function (k,m) { leaked_key = k; return hmac(k,m); };
```

$\rightarrow$ il faut réimplémenter la crypto dans la fonction g.

Appels de fonctions externes

```
...
var key = "...";
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://logging.example.com", false);
xhr.send(Crypto.HMAC(key, msg) + "," + msg);
...
```

Le script attaquant peut redéfinir la méthode `open` de `XMLHttpRequest` pour inspecter la pile et récupérer le code de la fonction `g` :

```
stackwalk = function() {
  var apisource = stackwalk.caller.toSource(); ...
}
```

Un code protégé

Séparer la fonction qui contient la clé et fait les calculs de la fonction qui communique.

```
var f_internal = (function (){
    var hmac = function(key,msg){ /* réimplémentation de HMAC */ }
    var g = function(msg){
        var key = "...";
        return (hmac(key,msg) + "," + msg);
    }
    return function(msg){return g(msg);}
})();
var f = function (msg) {
    var mac = f_internal(msg);
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://logging.example.com", false);
    xhr.send(mac);
}
```

Defensive JavaScript

(K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, *op. cit.*)

Un sous-ensemble de JavaScript, vérifié par typage statique :

- Portée lexicale stricte.
- Typage statique strict.
  Les objets et les tableaux sont initialisés à la création.
- Pas de coercions (`toString`) ni de *getters*/*setters* implicites.
- Séparation des tas mémoire : pas de partage d'objets entre le code défensif et l'extérieur.
  $\rightarrow$ Les points d'entrée dans le code défensif utilisent uniquement des types de base, p.ex. `string` $\rightarrow$ `string`.

Bilan sur la sécurité de JavaScript

Pas conçu pour la sécurité.

Politique de la même origine inopérante.

Un trait du langage (la portée statique) fournit des garanties d'isolation ; beaucoup d'autres traits la détruisent.

Machines à capacités

Deux visions qui convergent

Segmentation à grain fin :

- Chaque tableau, *record*, etc, réside dans un segment distinct.
- Une référence = un segment ;
  un pointeur = une paire (segment, déplacement).
- Contrôle automatique des bornes lors des accès
  $\rightarrow$ plus de *buffer overflow* !

Capacités (*capabilities*) pour accéder à la mémoire :

- Référence / pointeur = droit d'accéder à une zone mémoire.
- Pas d'accès possible (lecture, écriture, exécution)
  si le programme ne détient pas la capacité correspondante.

Vision abstraite d'une capacité mémoire

$$\text{capacité} = \langle \overbrace{\text{permissions}}^{\text{R, W, X, etc}}, \text{base}, \text{taille}, \text{décalage} \rangle$$

Donne des permissions sur les adresses $[\text{base}, \text{base} + \text{taille}[$

Lecture d'un mot de 32 bits :

$\texttt{load32}(\langle p, b, t, d \rangle) =$
si $\mathtt{R} \in p \wedge d + 4 \leq t$
alors lire 4 octets à l'adresse $b + d$
sinon erreur

Vision abstraite d'une capacité mémoire

Arithmétique de pointeurs :

$$\texttt{offset}(\langle p, b, t, d\rangle,\ \delta) =$$
$$\quad \text{si } \{\texttt{R}, \texttt{W}, \texttt{X}\} \cap p \neq \emptyset \wedge 0 \leq d + \delta \leq t$$
$$\quad \text{alors } \langle p, b, t, d + \delta\rangle \text{ sinon erreur}$$

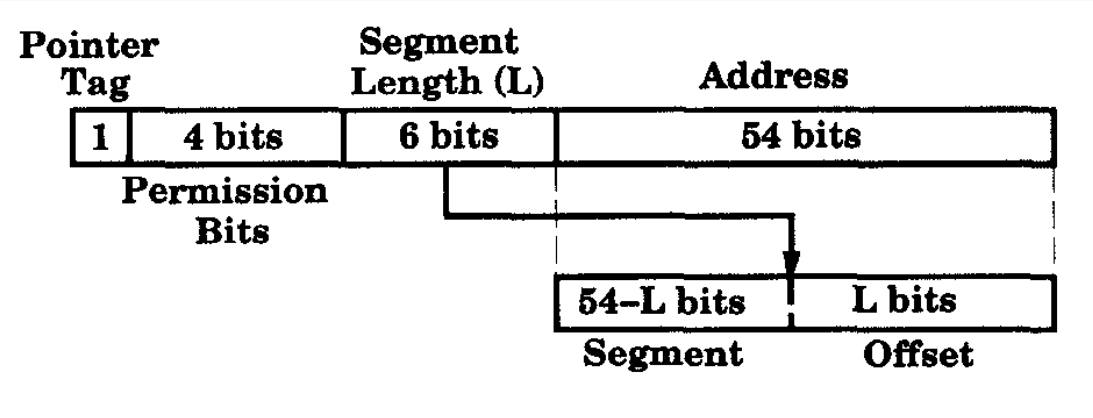
Restriction d'une capacité en taille et en permissions :

$$\texttt{subseg}(\langle p, b, t, d\rangle,\ p',\ t'\ ) =$$
$$\quad \text{si } p' \subseteq p \wedge d + t' \leq t$$
$$\quad \text{alors } \langle p', b + d, t', 0\rangle \text{ sinon erreur}$$

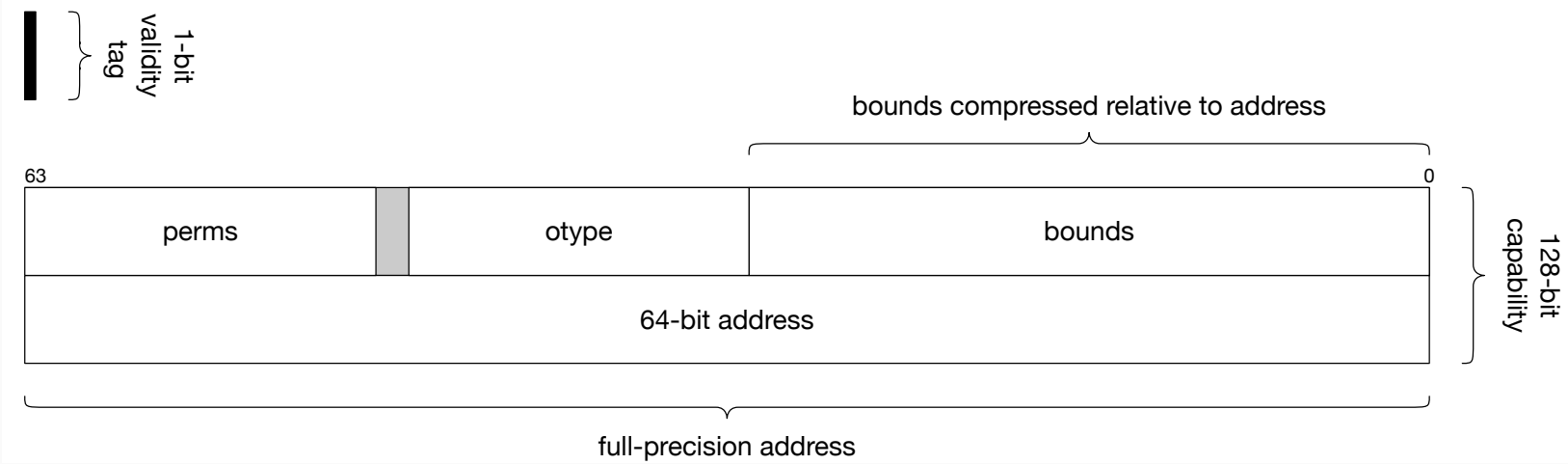
Propriété de *décroissance* : en mode utilisateur, un programme ne peut pas construire de capacités plus fortes que les capacités initialement accessibles depuis les registres et la mémoire.

Représentation concrète d'une capacité mémoire

Guarded pointers (Carter, Keckler, Dally, 1994) :



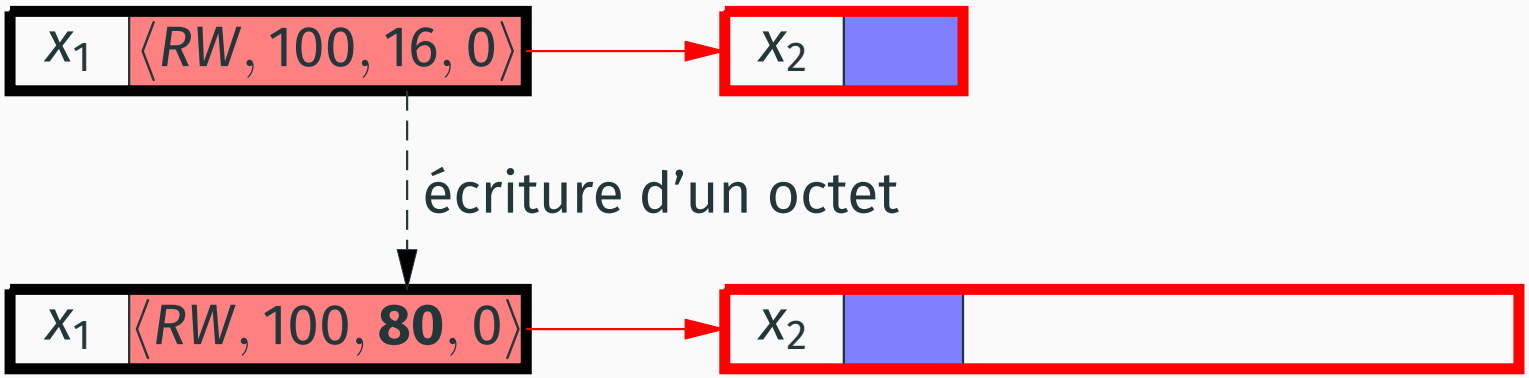
L'architecture CHERI (U. Cambridge, 2016) :



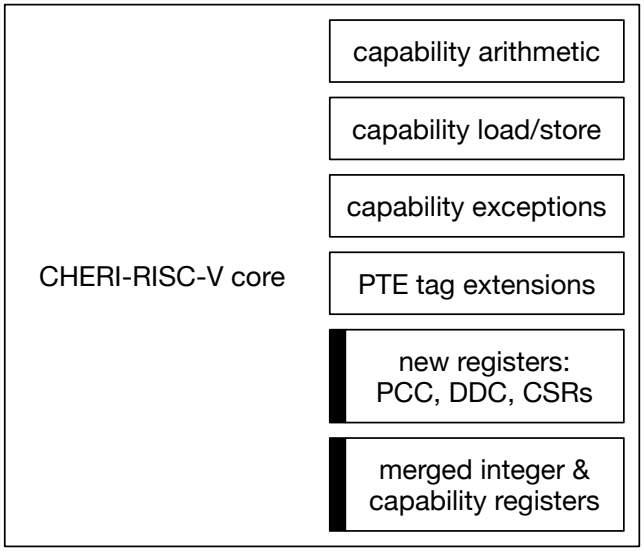
Intégrité des capacités

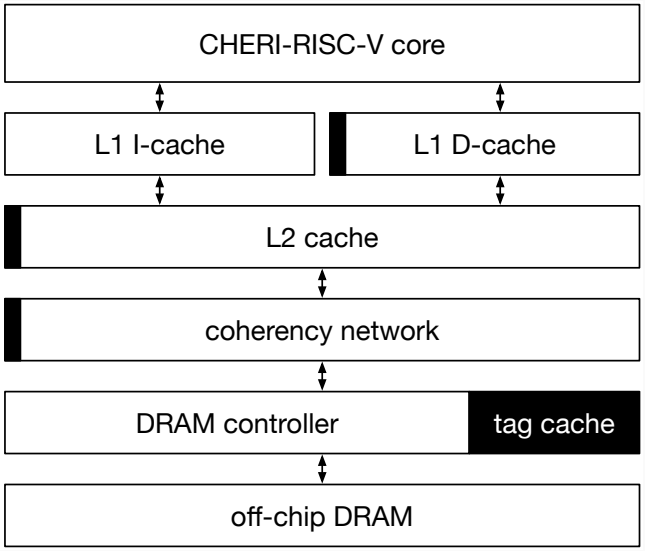
Tout comme les pointeurs, les capacités doivent pouvoir être stockées en mémoire, p.ex. pour faire des listes chaînées :

Qu'est-ce qui empêche le programme de fabriquer une fausse capacité en écrivant des octets dans la mémoire ?



Un *tag* pour les capacités valides





Ajouter un bit de plus (65^{e} ou 129^{e} bit) à chaque registre et à chaque mot mémoire. Ce bit de *tag* dit si une capacité est valide.

- Tag mis à 1 par les instructions qui produisent des capacités (`offset`, `subseg`, etc);
- Tag mis à 0 par toutes les autres instructions (arithmétique, logique, écriture d'un sous-mot, etc).

Protéger les données avec des capacités

Chaque variable, chaque bloc alloué dynamiquement (`malloc`) peut être placé dans «sa» zone mémoire, disjointe des autres, avec vérification automatique des bornes lors de tous les accès.

```
int f() {
    char b[80];                    // b = ⟨RW, 1000, 80, 0⟩
    int ok = 0;                    // &ok = ⟨RW, 1080, 4, 0⟩
    char * p = malloc(1024);       // p = ⟨RW, 4000, 1024, 0⟩
    gets(b);
    ...
}
```

En particulier, aucune affectation `b[i] = ...` ne peut modifier `ok`, même si les zones mémoires correspondantes sont adjacentes

$\rightarrow$ l'appel `gets(b)` est sûr!

Programmer son allocateur mémoire

```
#define HEAPSIZE 1000000
static char heap[HEAPSIZE];
static int next = 0;

void * malloc(size_t sz) {
    if (next + sz >= HEAPSIZE) return NULL;
    void * p = subseg(heap + next, RW, sz);
    next += sz;
    return p;
}
```

Le pointeur renvoyé par `malloc` permet uniquement d'accéder aux cases $\texttt{next}, \ldots, \texttt{next} + \texttt{sz} - 1$ du tableau `heap`.

Pour rendre le bloc alloué inaccessible autrement que via ce pointeur, il suffit de rendre le tableau `heap` invisible aux clients.

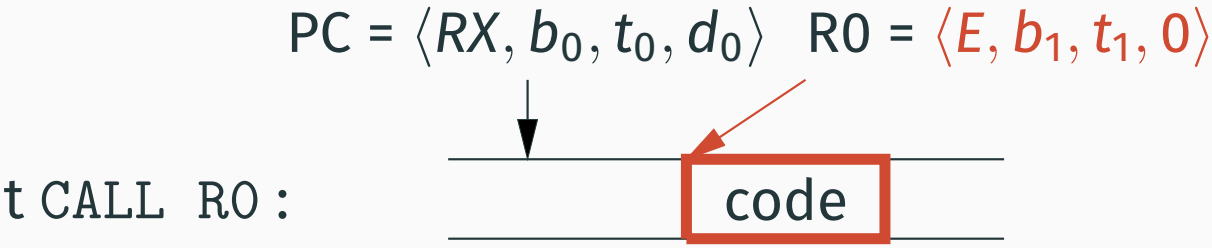
Protéger le code avec des capacités

En utilisant `subseg` on peut sceller un pointeur de code, faisant passer ses capacités de `RX` à `E` (comme `Enter`).

Un pointeur scellé `E` ne donne aucun accès à la zone mémoire, mais permet juste à une instruction `CALL` de sauter à l'adresse correspondante.

Le code appelé récupère les permissions `RX` sur la zone mémoire.

Avant `CALL R0` :



Pas d'arithmétique de pointeurs sur les capacités `E`
$\rightarrow$ pas de saut «au milieu» du code d'une fonction !

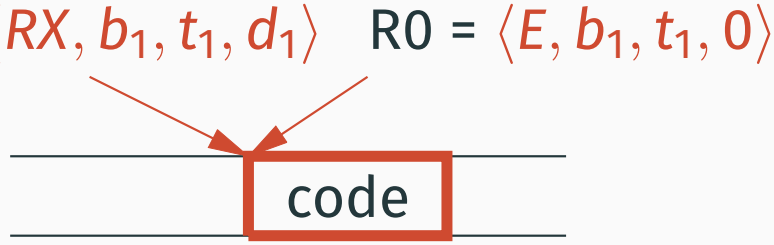
Protéger le code avec des capacités

En utilisant `subseg` on peut sceller un pointeur de code, faisant passer ses capacités de `RX` à `E` (comme `Enter`).

Un pointeur scellé `E` ne donne aucun accès à la zone mémoire, mais permet juste à une instruction `CALL` de sauter à l'adresse correspondante.

Le code appelé récupère les permissions `RX` sur la zone mémoire.

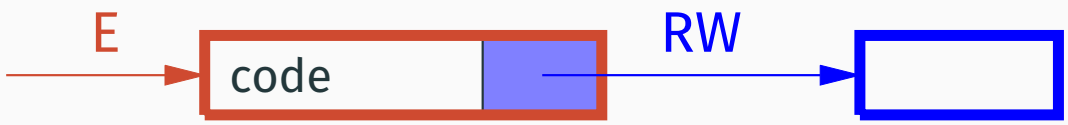
Après `CALL R0` :



Pas d'arithmétique de pointeurs sur les capacités `E`
$\rightarrow$ pas de saut «au milieu» du code d'une fonction !

Données privées à un morceau de code

La zone de code scellée peut contenir une capacité `RW` vers une zone de données. Si cette zone de données n'est pas visible de l'appelant, elle est donc privée au code scellé.



Données privées à un morceau de code

Exemple : un compteur monotone (en pseudo-assembleur).

```
    // Zone de code RX                    // Zone de données RW
next: r1 = load(data)                    counter: .word 0
      r0 = load(r1)
      if r0 = MAX_INT: abort
      r0 = r0 + 1
      store(r1, r0)
      r1 = 0      // attention aux fuites!
      return r0
data: .word counter
```

Un client auquel on donne uniquement une capacité E sur `next` ne peut pas modifier `counter` directement, mais uniquement en appelant la fonction `next`.

Encapsulation procédurale ou par objets

Ce genre d'encapsulation d'une donnée dans du code s'exprime dans beaucoup de langages de programmation en utilisant la *portée lexicale* des liaisons de variables.

```
int next(void) { static int counter = 0; return ++counter; }

let next =
  let counter = ref 0 in fun () -> incr counter; !counter

class Counter {
  private int counter = 0;
  public int next() { return ++counter; }
}
```

Les capacités peuvent garantir cette encapsulation même si l'attaquant ne respecte pas la portée lexicale.

Bilan sur les architectures à capacités

Un espoir : une isolation légère et à grain fin (comme du logiciel) mais fiable (comme du matériel).

Une idée déjà ancienne (Dennis & Van Horn, 1966).

Un échec commercial (Intel iAPX 432, 1981).

Un regain d'intérêt avec l'architecture CHERI (U. Cambridge, 2016) et le processeur ARM Morello (2022).

Question encore ouverte : quelles sont les capacités nécessaires pour isoler des langages de haut niveau ?

(p.ex. A. Lippeveldts (2019) propose des capacités linéaires pour isoler la pile.)

Liens intéressants avec la logique de séparation.

(Georges et al, *Cerise : program verification on a capability machine in the presence of untrusted code*, preprint, 2021.)