

OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



Mise à plat de graphes de flot de
contrôle et exécution symbolique

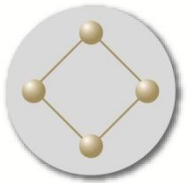
SSTIC 2013



L'auteur

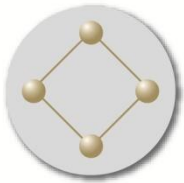


- Eloi Vanderbeken
 - @elvanderb
 - Ingénieur SSI
 - Oppida : Conseil et expertise en sécurité des systèmes d'information
- Reverse-engineering
- Cryptographie appliquée
- Obfusca... brouillage !



Plan

- ■ Introduction
- Approche proposée
- Implémentation pratique
- Résultats

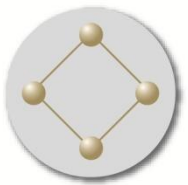


Introduction

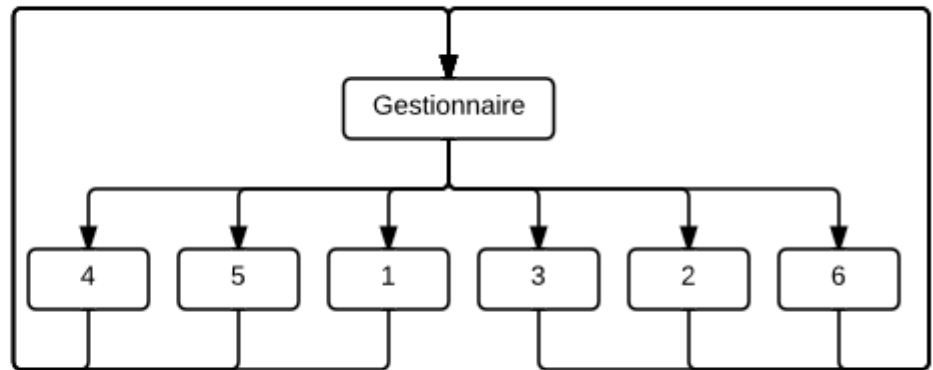
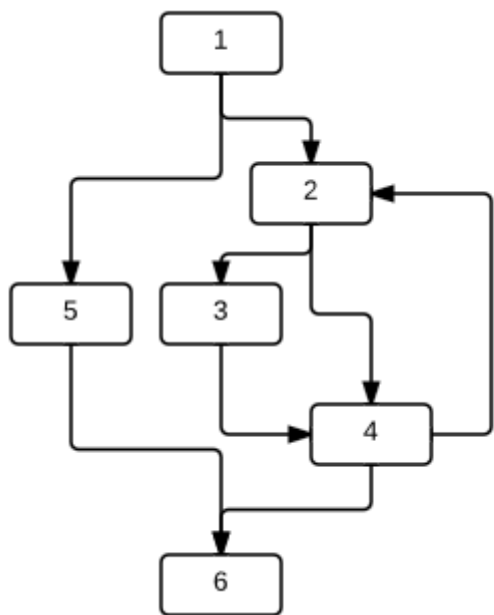
- Plusieurs niveaux de brouillage de code
- Techniques *bas niveau* :
 - Agissent directement sur le code assembleur.
 - Pas ou peu d'informations contextuelles.
 - Ex : insertion de code mort, couches de chiffrement, mutation / virtualisation de code, etc.
 - Méthodes générique de débrouillage (optimisation du code, approche concolique etc.)
- Techniques *haut niveau* :
 - Nécessitent des informations contextuelles :
 - Etats de variables globales ;
 - Graphe de flot de contrôle ;
 - Effets de bord des fonctions ;
 - etc.
 - Problèmes indécidables ;
 - Ex : mise a plat de graphe de flot de contrôle, prédicats opaques utilisant des variables globales, etc.
 - Pas de méthodes génériques.

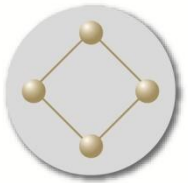
Mise à plat de graphe de flot de contrôle

- ■ Méthode décrite formellement par Wang en 2000.
- Les relations entre les blocs ne sont pas directement visibles.
- Un (ou plusieurs) gestionnaire(s) se charge(nt) d'aiguiller le flot d'exécution.
- L'aiguillage est fait à l'aide de variables initialisées / modifiées par le code précédemment exécuté.

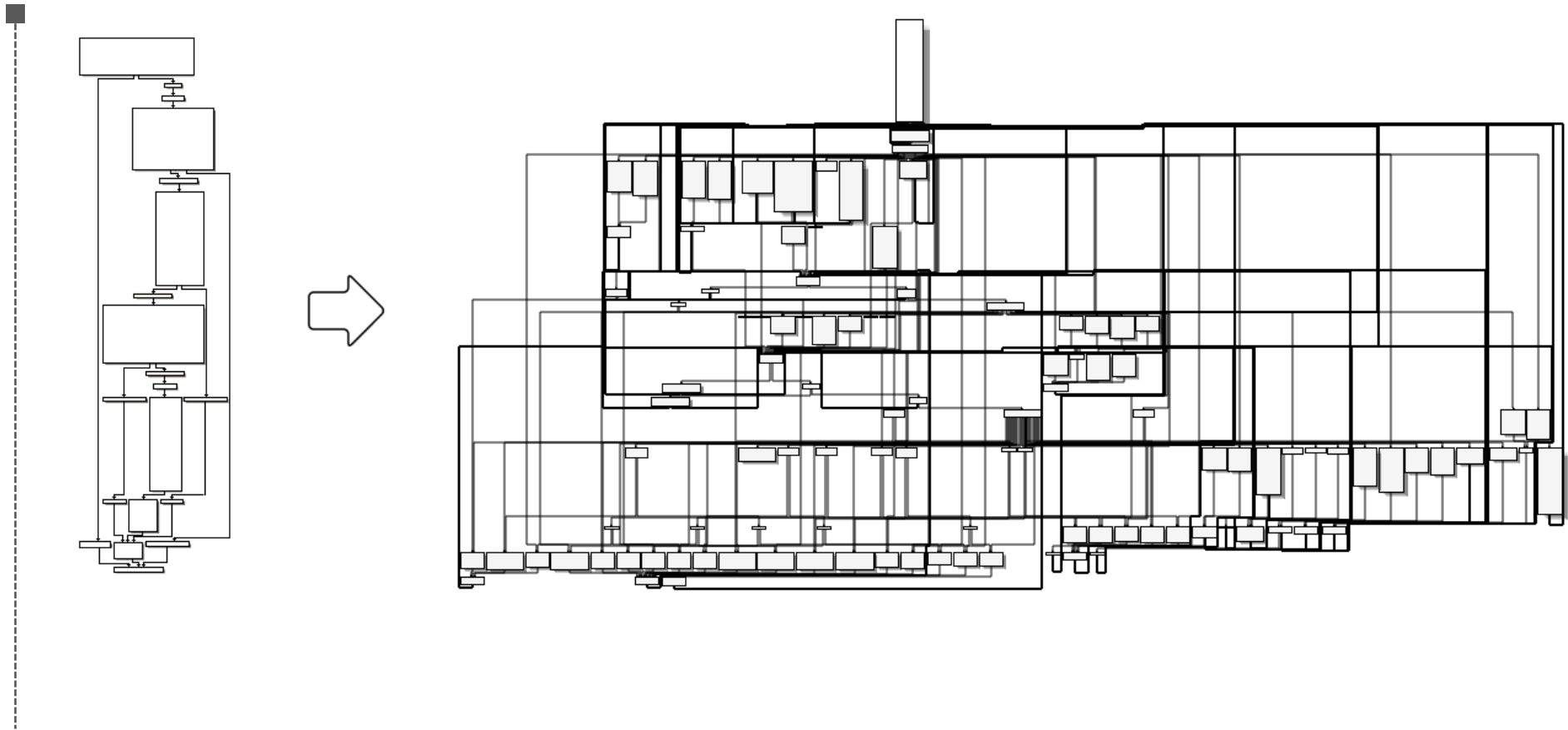


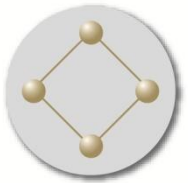
Mise à plat de flot de contrôle en une image





Mise à plat de flot de contrôle en une image (ou deux)





Approches possibles ...

- Dynamique avec suivi d'exécution
- Dynamique avec suivi de donnée
- Statique avec de l'exécution symbolique.

Approche dynamique avec suivit d'exécution

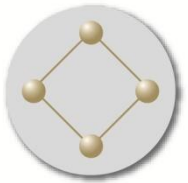
- Principe :
 - Exécution de la fonction brouillée ;
 - Enregistrement de l'enchaînement des basic blocks ;
 - Reconstruction du CFG original.
- Avantages :
 - Rapide ;
 - Précis ;
 - « Simple »
- Inconvénients :
 - Nécessite d'identifier les gestionnaires ;
 - Ne donne pas les conditions de branches ;
 - N'est pas exhaustif.

Approche dynamique avec suivi de donnée

- Principe :
 - Exécution de la fonction brouillée ;
 - Enregistrement des opérations sur les données ;
 - Le flot d'exécution est ignoré.
- Avantages :
 - Précis ;
 - *Relativement* simple.
- Inconvénients :
 - Ne donne pas la structure du code ;
 - N'est toujours pas exhaustif ;
 - Peu efficace en cas d'opérations mathématiques complexes.

Approche choisie – l'approche statique

- Principe :
 - Considération de l'ensemble des entrées possibles ;
 - Suivi du flot d'exécution de la fonction à partir de ces entrées ;
 - Reconstruction du CFG original à partir de ces informations.
- Avantages :
 - Exhaustif ;
 - Permet d'avoir une vision plus abstraite du code ;
 - Fournit les conditions de branche ;
 - Élégant 😊.
- Inconvénients :
 - Parfois trop abstrait :
 - quelle est la valeur de PC après l'instruction `ADD PC, R0, [SP + Sym]` ?
 - Complexe et lourd.



Quelques notions ...

- Traces
- Basic Blocks
- Exécution symbolique

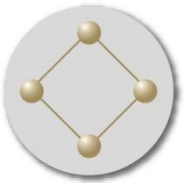
Trace

- ■ Proche de la notion de trace de Pin :
 - une entrée ;
 - plusieurs sorties ;
 - composée de Basic Blocks (BBI);
 - pas de boucle.

- Mais différentes :
 - Les traces sont des arbres.
 - En cas de branche conditionnelle, la trace contient les deux branches.
 - Une trace est terminée par un gestionnaire.

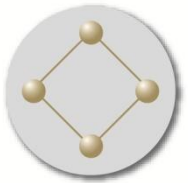
Basic Block

- ■ Différents des Bbl « classiques ».
- Suite d'instructions à une entrée et une sortie.
- Un bloc de base est identifié par son adresse de début **et par son parent**.
- Caractérisé par son contexte initial.
- Peut contenir des instructions de branches conditionnelles :
 - Si le contexte initial ne permet pas plusieurs flux d'exécution différents.



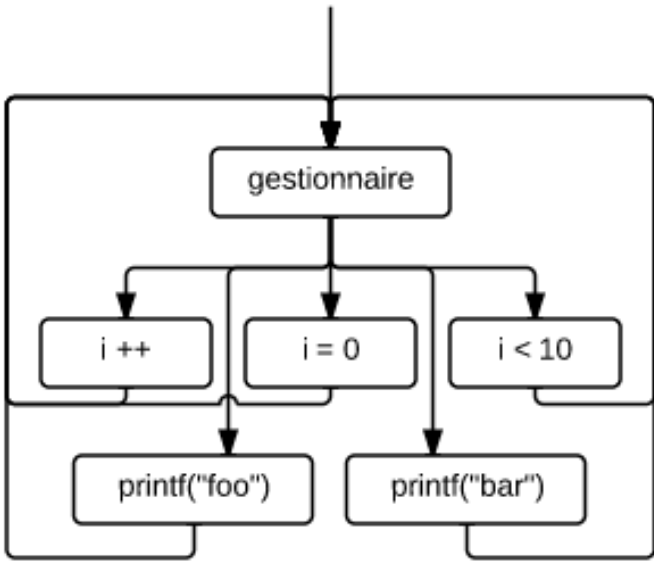
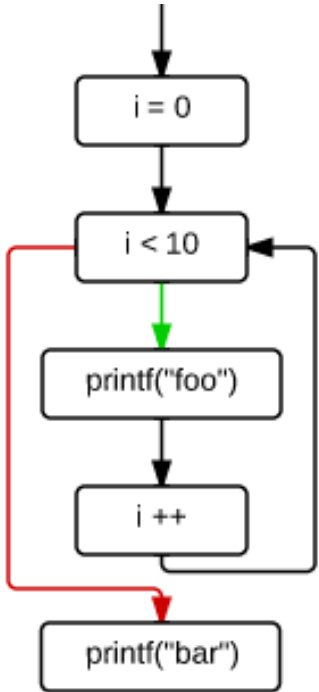
Principe de reconstruction du graphe

- Exécution symbolique du code.
- Si instruction conditionnelle dans le contexte donné :
 - Fin de *Basic Block*.
- Si instruction de fin de gestionnaire :
 - Fin de *Trace*.
- Si instruction précédemment exécutée :
 - Fusion des contextes.
- Sinon ajout de l'instruction au *Basic Block* courant.

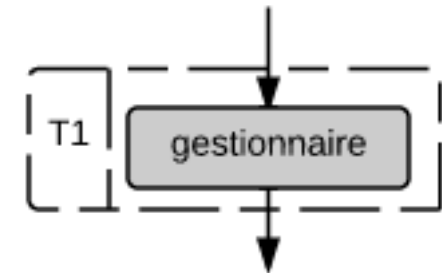
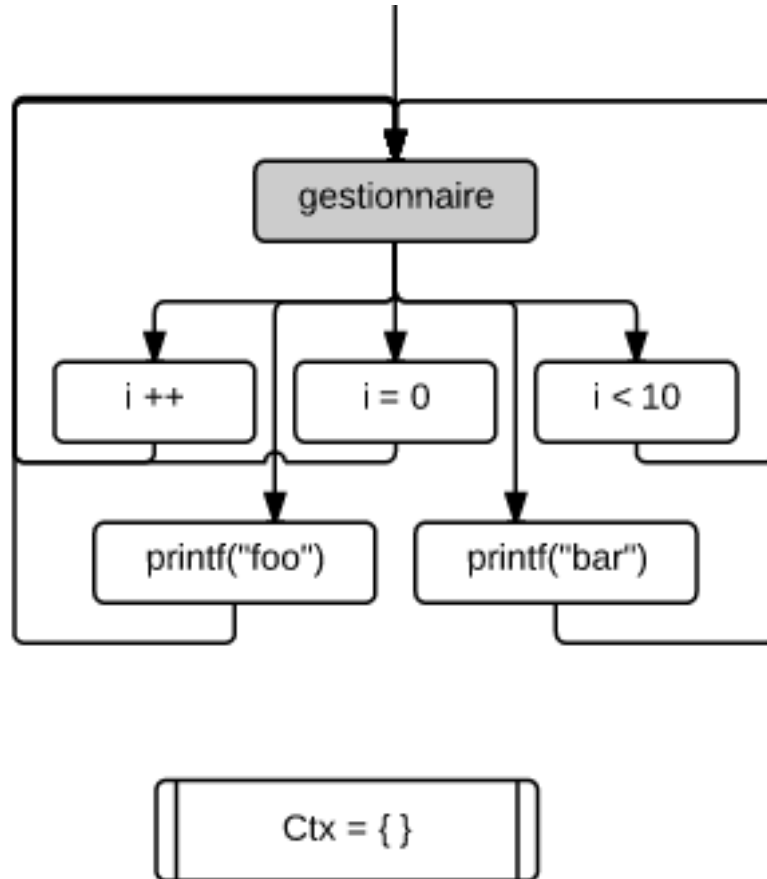


Un exemple en image

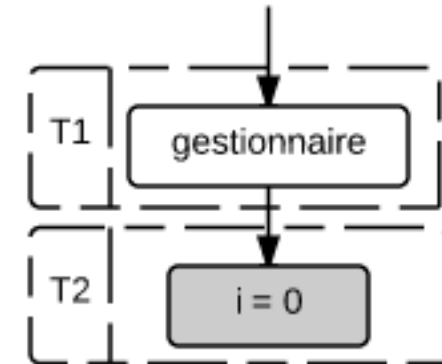
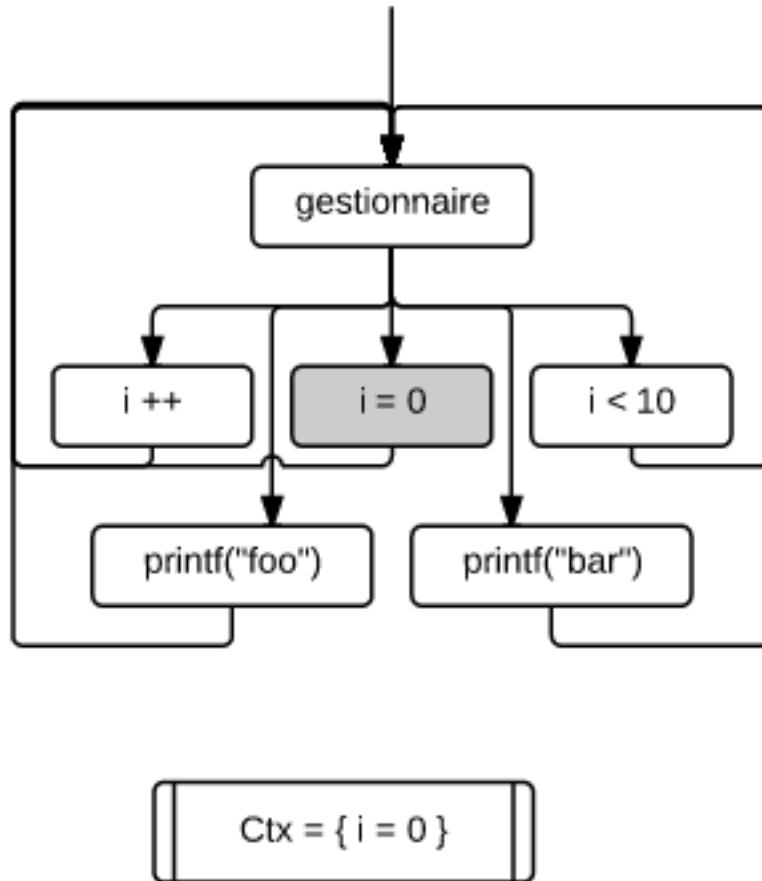
```
for (i = 0; i < 10; i++)  
  printf("foo\n");  
  printf("bar");
```



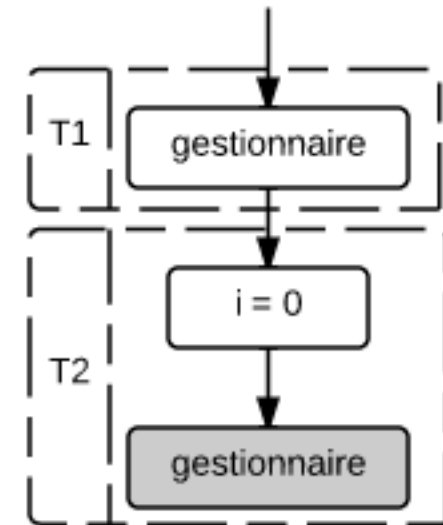
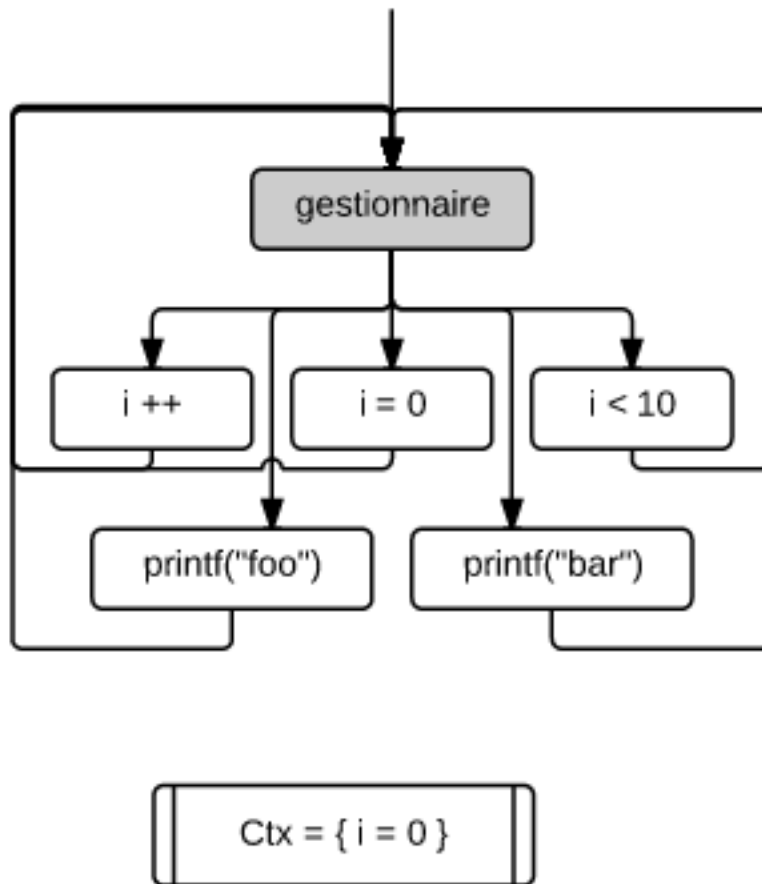
Un exemple en image



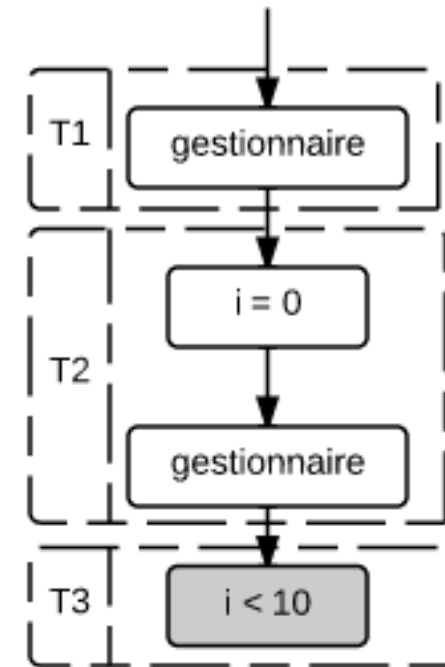
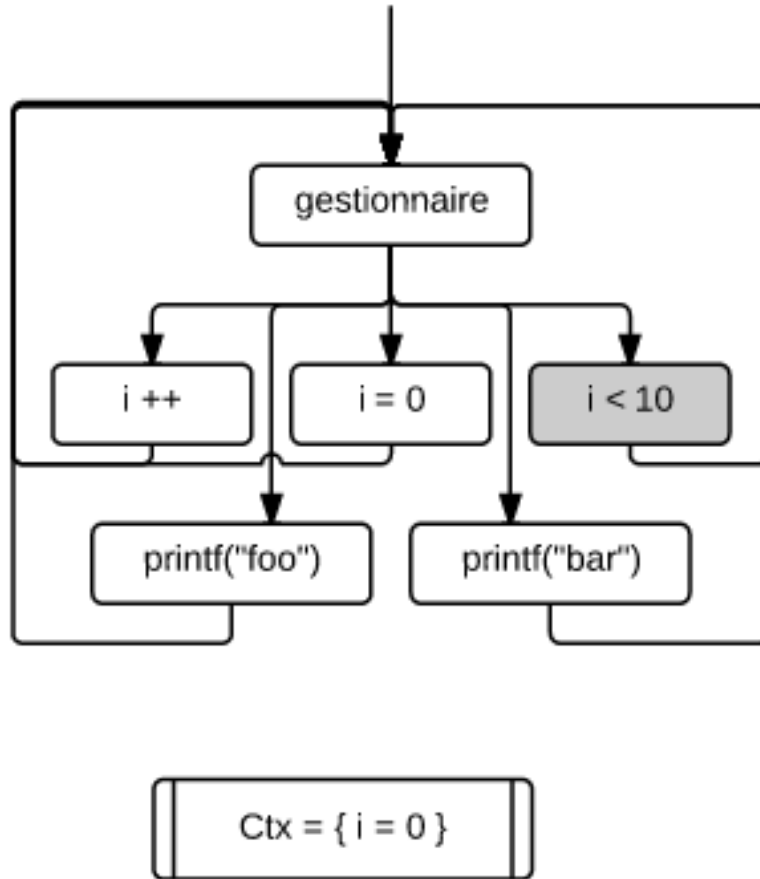
Un exemple en image

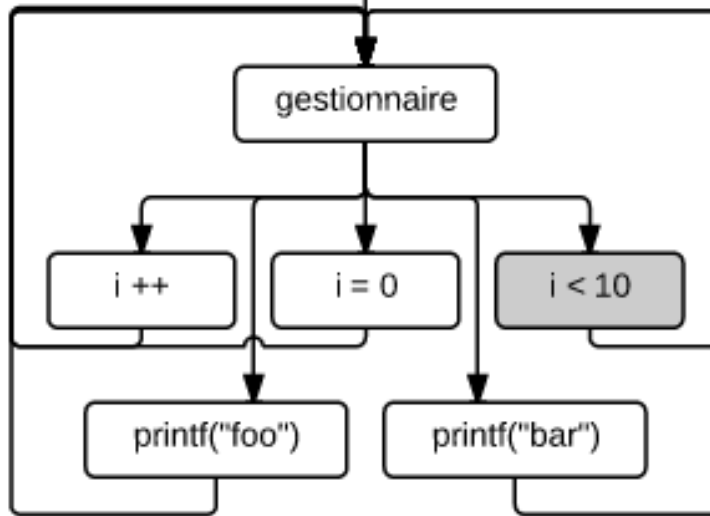
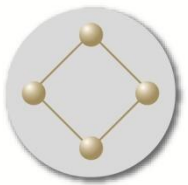


Un exemple en image

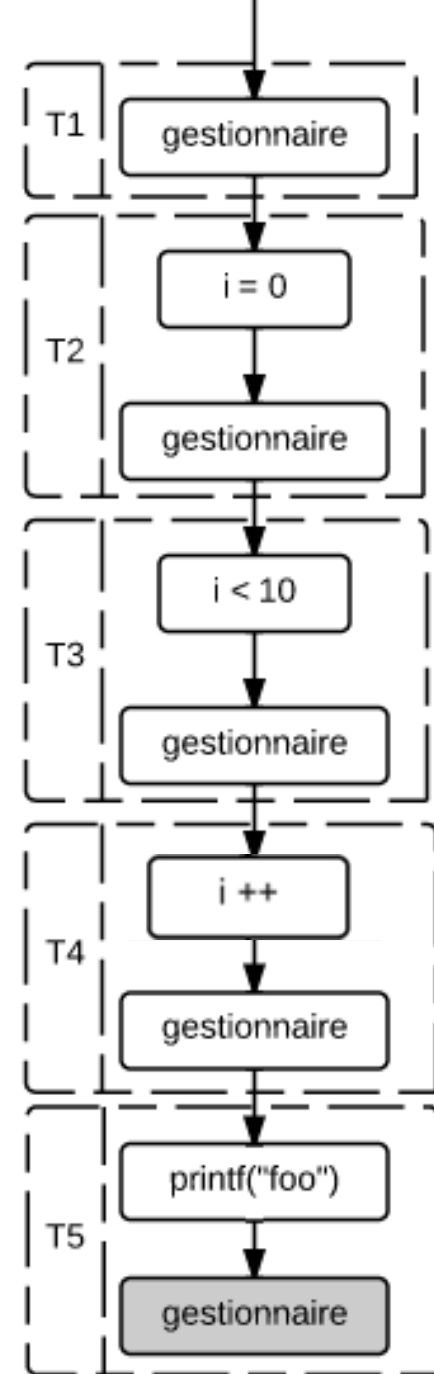


Un exemple en image

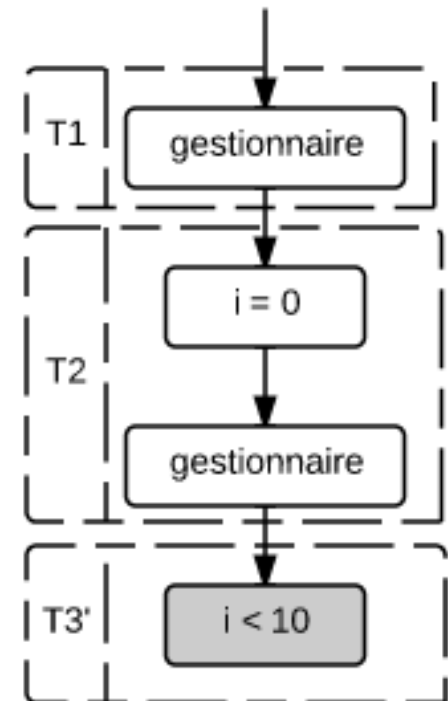
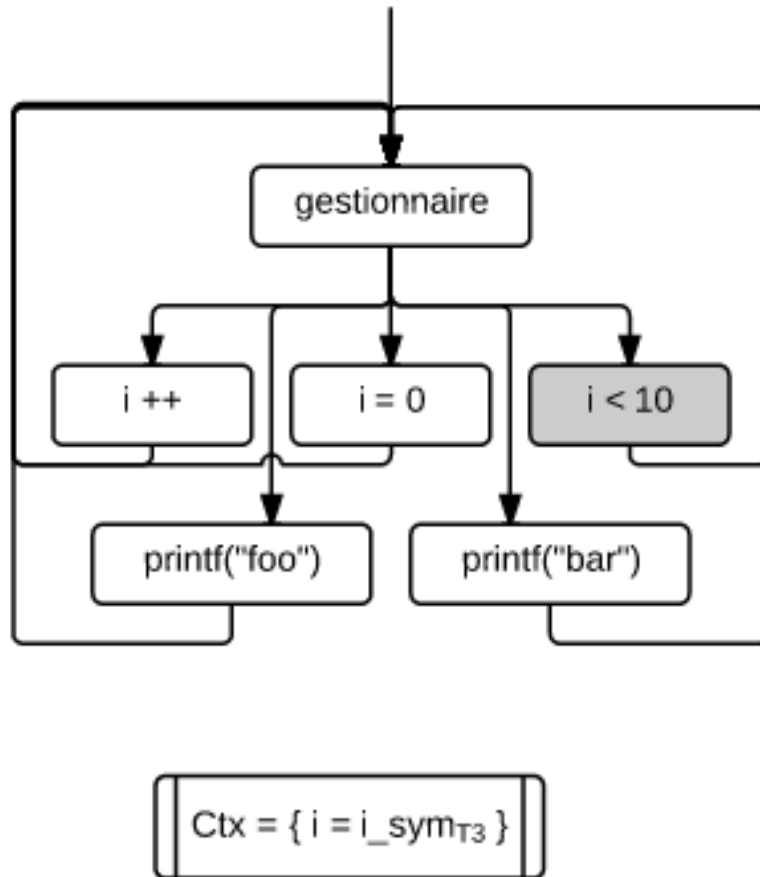


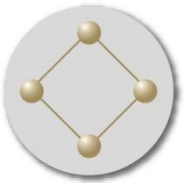


Ctx = { i = 1 }



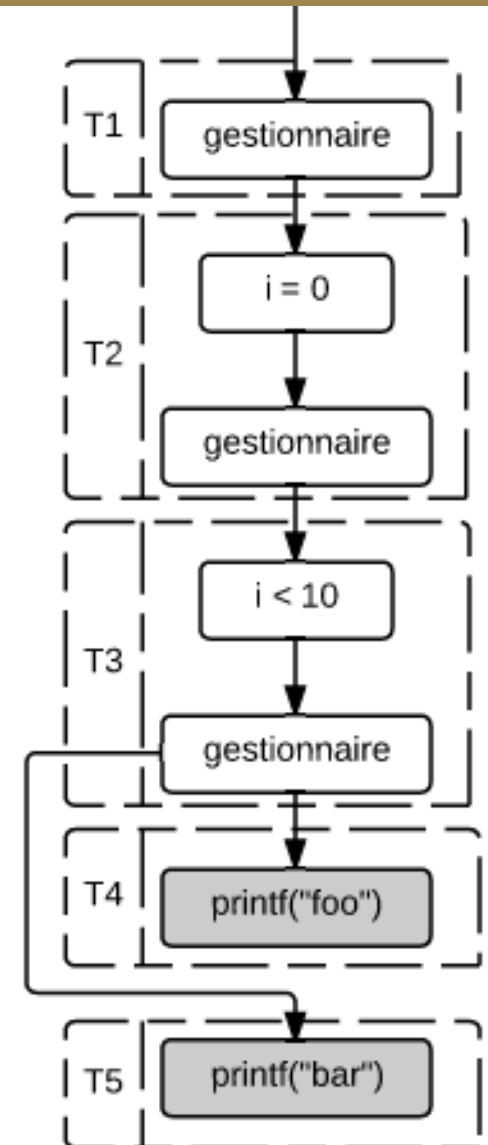
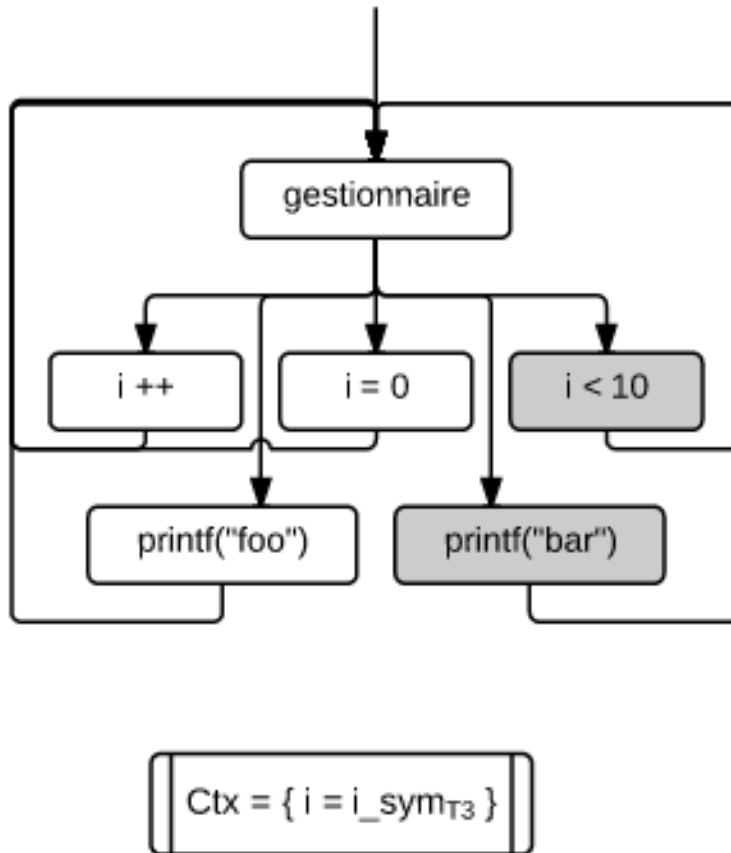
Un exemple en image

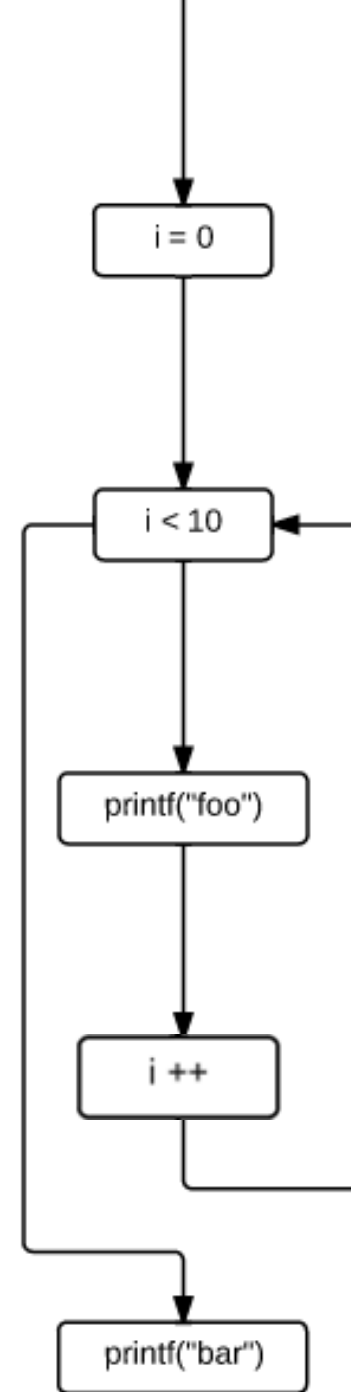
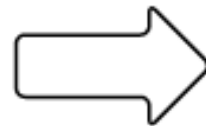
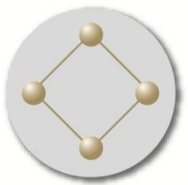




Un exemple en image

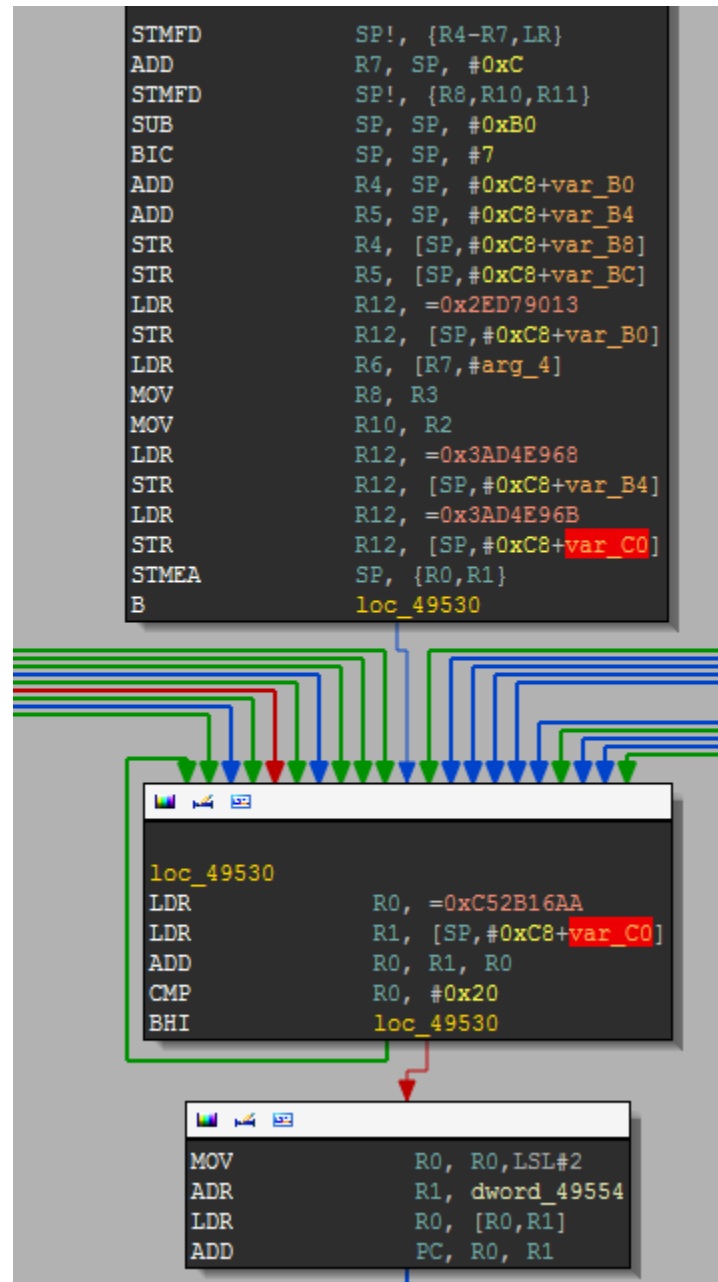
OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION

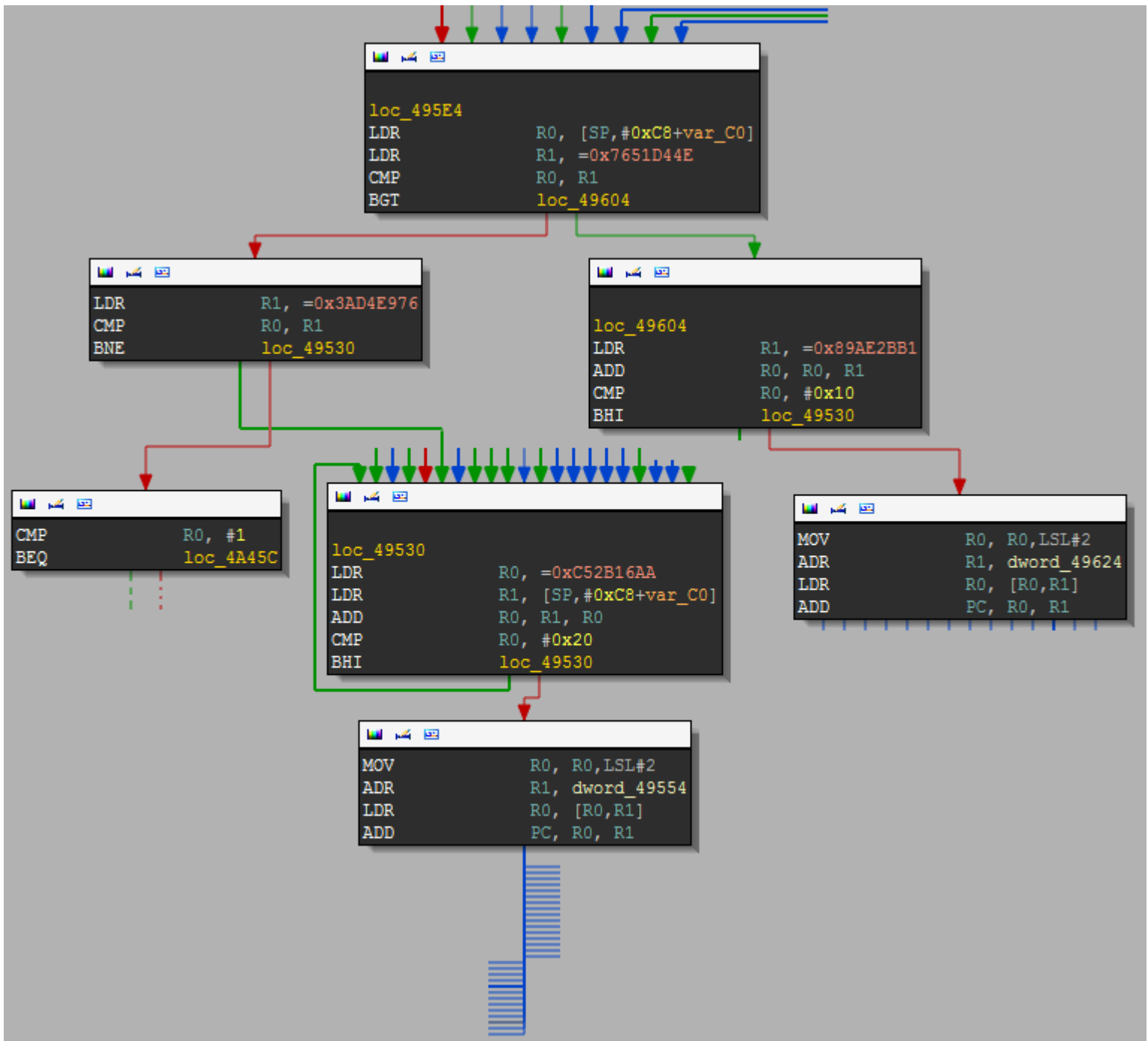
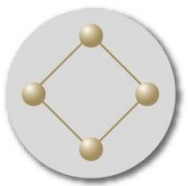




Implémentation concrète

- Gestion des références à des adresses symboliques ?
- Gestion des appels aux fonctions externes ?
- Simplification / résolution des équations ?
- Simplification du code ?
- Identification des gestionnaires ?
- Extraction de la sémantique des instructions ?





Traduction IDA → Objet python

- ■ Pas si simple qu'il n'y paraît :
 - Documentation pauvre voir inexistante.
 - Réutilisation de champs prévus pour d'autres architectures.
 - Condition d'exécution dans le champ `segpref`
- Instruction résumée à l'aide de 3 types d'objets :
 - Opcode (`MOV, OR, B` etc.)
 - Condition d'exécution (`NZ, CC, GE` etc.)
 - Opérande(s) (`R0, 0x1234, R0 LSL R3, [SP!, 0x456]` etc.)
- Description de la sémantique des instructions à l'aide de ces objets.
 - Simple et rapide.
 - N'a à être effectuée qu'une seule fois.

Exemple : implémentation d'une instruction

```
class Bic(Ins) :
    __metaclass__ = MetaIns

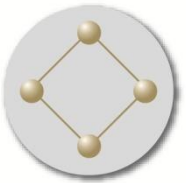
    def _emu(self, context) :
        # les conditions d'exécution sont vérifiées par Ins.emu
        if len(self.ops) != 3 :
            raise EmulationException("%s is not supported yet"%self)
        dest = self.ops[0]
        src1 = self.ops[1]
        src2 = self.ops[2]
        v = context[src1] & ~context[src2]
        context[dest] = v
        if self.affect_flags :
            context.flags.N = v.N
            context.flags.Z = v.Z
```

Contexte symbolique

- ■ Basé sur Z3
 - solveur d'équations SAT développé par Microsoft ;
 - supporte les vecteurs de bits et l'arithmétique associée ;
 - performant ;
 - possède une interface python.
- Les registres et cases mémoire sont des valeurs de 32 bits.
- Les flags sont des booléens.
- Pile et reste de la mémoire dissociés
 - SP est une valeur symbolique.
 - Imprécision :
 - Une adresse absolue pourrait correspondre à la pile ;
 - Une adresse relative à SP pourrait correspondre à une variable globale ;
 - En pratique ce n'est jamais le cas : imprécision acceptable.

Contexte symbolique – accès mémoire en lecture

- ■ Si l'adresse à déréférencer est concrète :
 - Mémoire globale ;
 - Lecture de la valeur dans la base de IDA.
- Si l'adresse est symbolique :
 - Détermination des valeurs possibles de l'adresse à l'aide de Z3 ;
 - Si SP intervient dans le calcul de l'adresse :
 - Il s'agit d'une adresse dans la pile ;
 - Soustraction de SP à l'adresse et calcul des deltas possibles.
 - Si plus de N valeurs possibles, abstraction de la valeur ;
 - Sinon construction de la valeur déréférencée :
 - `If (addr_sym == addr_concrete1, Dword(addr_concrete1),`
`If (addr_sym == addr_concrete2, stack[addr_concrete2-SP],`
`..., ∅)`

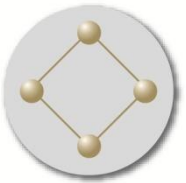


Exemples de dérérérencement

```
a = arg0 & 1; /* a = arg0 & 1 */  
b = "ab"[a];  
/* b = If(arg0 & 1 == 0, 'a', If(arg0 & 1 == 1, 'b', Ø)) */  
  
c = ((DWORD*) arg0) [1];  
/* c = sym_deref_(arg0+4) */
```


Contexte symbolique – accès mémoires en écriture

- ■ Si l'adresse à déréférencer est concrète :
 - Mémoire globale ;
 - Interdit dans notre modèle :
 - Pourrait être implémenté facilement.
- Si l'adresse est symbolique :
 - Détermination des valeurs possibles de l'adresse ;
 - Si moins de N valeurs possibles :
 - `stack[addr_concrete1-SP] = If(addr_sym == addr_concrete1, new_value, orig_value1)`
 - `stack[addr_concrete2-SP] = If(addr_sym == addr_concrete2, new_value, orig_value2)`
 - etc.
 - Sinon ?



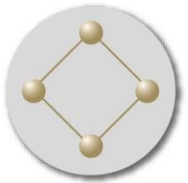
Exemples de dérérérencement

```
char t[4];  
a = arg0 & 3;  
/* a = arg0 & 3 */  
t[a] = 1;  
/* a = arg0 & 3 */  
/* t[0] = If(arg0 & 3 == 0, 1, sym_deref_(t)) */  
/* t[1] = If(arg0 & 3 == 1, 1, sym_deref_(t+1)) */  
/* t[2] = If(arg0 & 3 == 2, 1, sym_deref_(t+2)) */  
/* t[3] = If(arg0 & 3 == 3, 1, sym_deref_(t+3)) */  
arg0[arg1] = 3;  
/* ??? */
```

Contexte symbolique – écriture à une adresse symbolique

- ■ Nécessaire de déterminer quelles variables sont invariantes :
 - lors d'un appel à une fonction inconnue ;
 - lors d'une écriture mémoire à une adresse symbolique.
- Considérer toutes les variables invariantes :
 - Toutes les exécutions possibles ne seront pas considérées :
 - Toutes les branches ne seront pas nécessairement explorées.
 - `b = FALSE; f_set_bool(&b); if (b) { /* non exploré */ }`
- Considérer toutes les valeurs volatiles :
 - Sûre mais potentiellement *trop* abstrait :
 - `a = 0; f_foo(b); func_table[a](); // adresse de la fonction appelée ?`

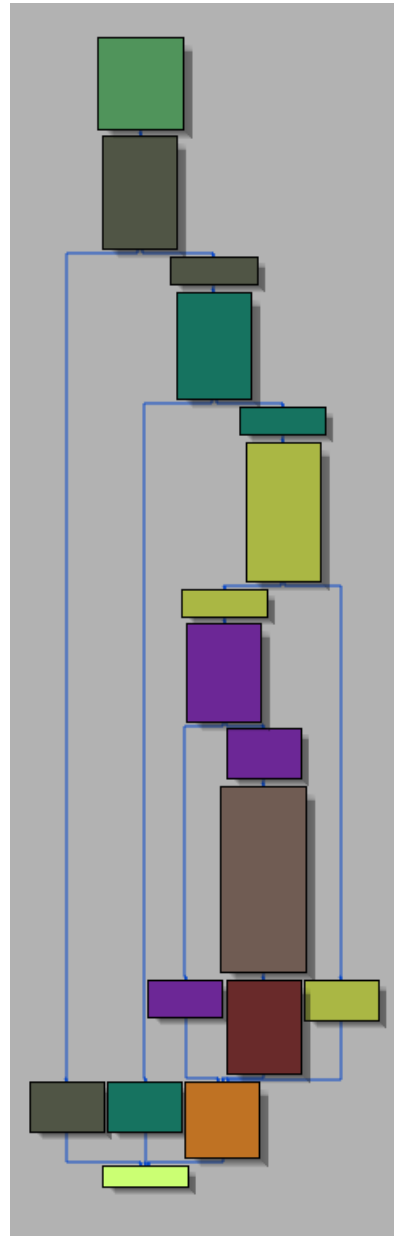
- ■ Définition « à la main » des invariants :
 - Sûre si l'analyse est correcte.
- Heuristique :
 - Seules les valeurs à haute entropie (identifiants des successeurs) et les pointeurs sur la pile sont gardés.
 - Les autres variables sont considérées comme volatiles.
 - Non sûre mais fonctionne en pratique.
- En pratique :
 - utilisation de l'heuristique et vérification à la main.



OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



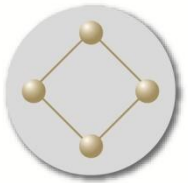
05/06/2013



Mise à plat de graphes de flot de contrôle et
exécution symbolique – SSTIC 2013

Simplification du CFG

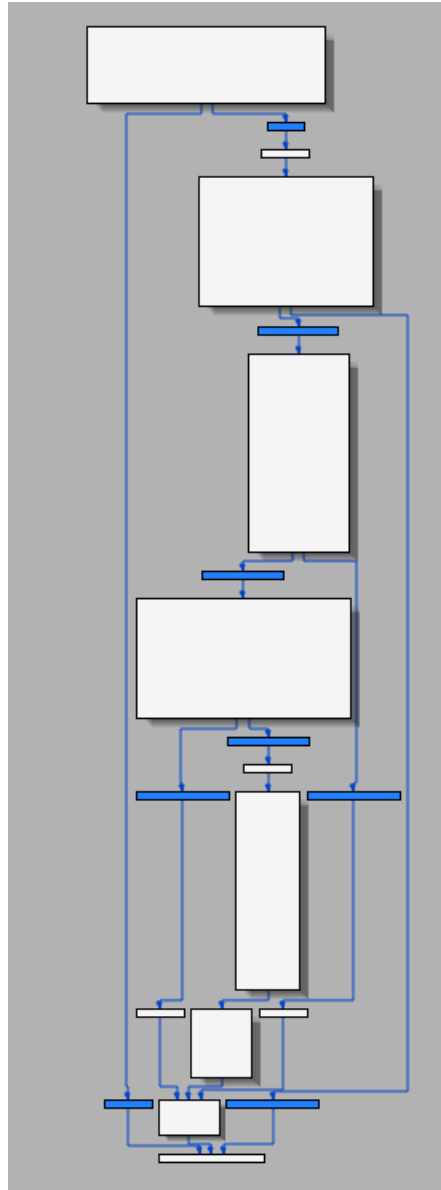
- - Exécution symbolique de toutes les traces.
 - Enregistrement de toutes les modifications :
 - Registres, mémoire.
 - Génération des conditions de saut :
 - simplification de `PC_sym == PC_concret`
 - Optimisation de la trace générée :
 - La simplification des formules est fournie *gratuitement* par Z3.
 - Les fonctions non connues sont supposées respecter la convention d'appel cdecl :
 - Toutes les variables locales sont considérées comme potentiellement utilisées.
 - Les registres `R0`, `R1`, `R2`, `R3` et `R12` sont considérés comme potentiellement utilisés.
 - Suppression des affectations inutiles :
 - Si une variable ou un registre `a` est initialisé et n'est pas utilisé avant sa prochaine initialisation alors l'initialisation est inutile.
 - Suppression des blocs vides.



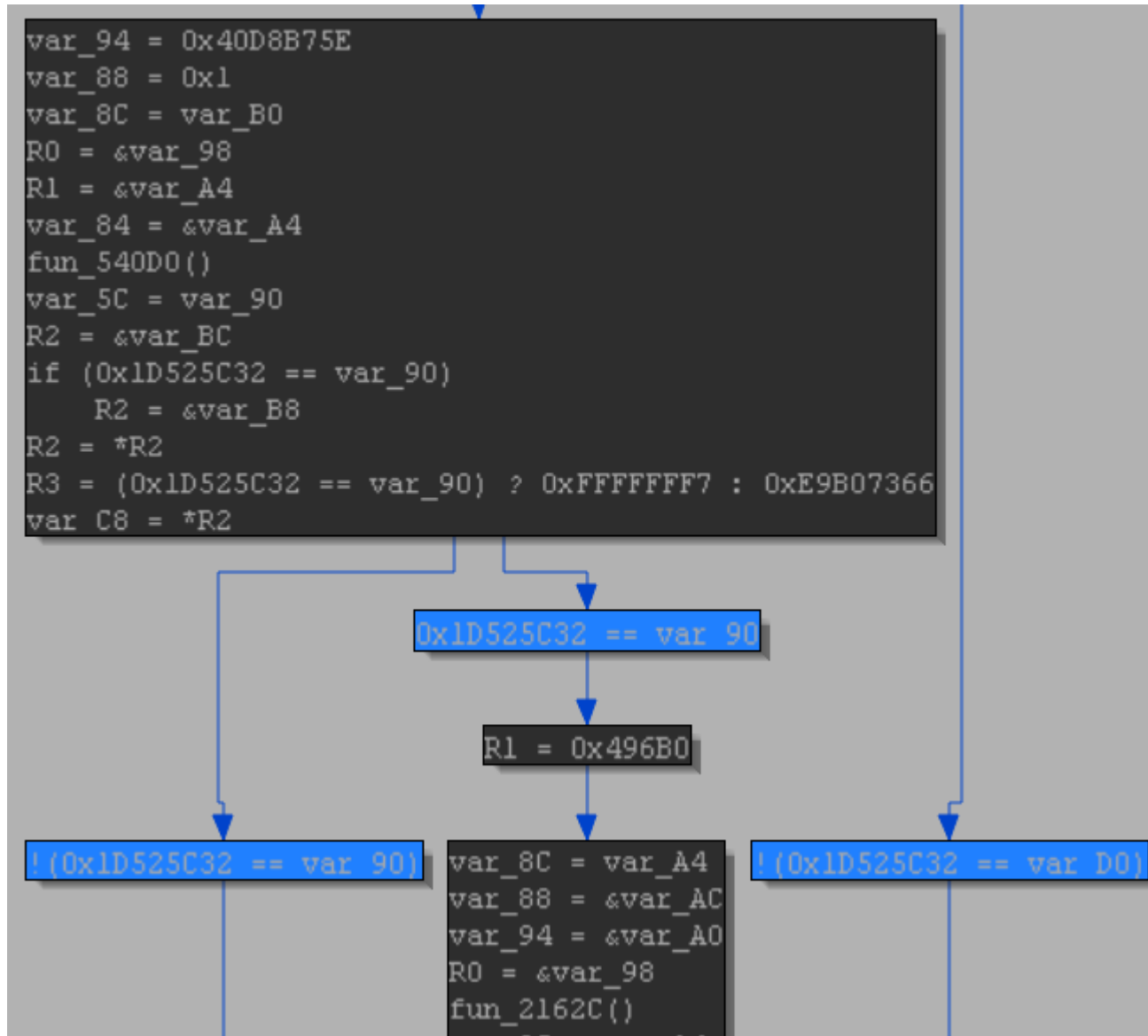
OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION

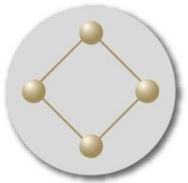


05/06/2013



Mise à plat de graphes de flot de contrôle et
exécution symbolique – SSTIC 2013

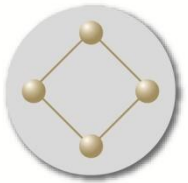




OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



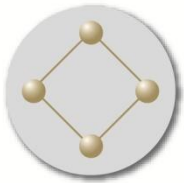
Démo



OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION

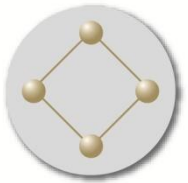


Conclusion



OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION

Merci de votre attention
(et à Jean Sigwald 😊)



OPPIDA
EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION

Questions ?