Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Obfuscator Workshop Final

Yverdon-les-Bains // 03-12-2012

RCSO-TIC
Réseau de Compétences de Suisse occidentale en technologies de l'information et de la communication

Agenda

- Accueil, présentation du contexte (30’)
- Obfuscation avec LLVM (45’)
- Pause (15’)
- Protection de binaires ARM (45’)
- Perspectives et conclusion (15’)
- Apéro

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Contexte

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Qui?

- HEIG-VD
  - Pascal Junod, Grégory Ruch, Julien Rinaldini
- EIA-FR
  - Jean-Roland Schuler, Adrien Giner, Marc Romanens

Quoi ?

- Etude des techniques de protection logicielle
  - Source code
  - Binaire
- Développement de prototypes d'outil de protection

Combien ?

- CHF 160’000, financés par la HES-SO via le RCSO-TIC
- 200 heures (professeurs)
- 2000 heures (assistants de recherche)
- Effort réparti équitablement entre les deux instituts

Quand ?

- Décembre 2010 à décembre 2012
- Prévu de terminer en décembre 2011...
- Difficultés à recruter des assistants

Comment?

- WP1: Etude de l'état de l'art, acquisition de compétences (2 mois)
- WP2{a,b}: Conception et développement d'un outil d'obfuscation, de «watermarking» et de «tamperproofing» de code
- WP3: Réalisation du rapport contenant les synthèses des différentes parties du projet

Pourquoi?

- Différents types d’adversaires:

Adversaires de type «black-box»

- Respectent les règles (!)
- Interagissent avec les composants logiciels en accord avec les interfaces définies

Adversaires de type «grey-box»

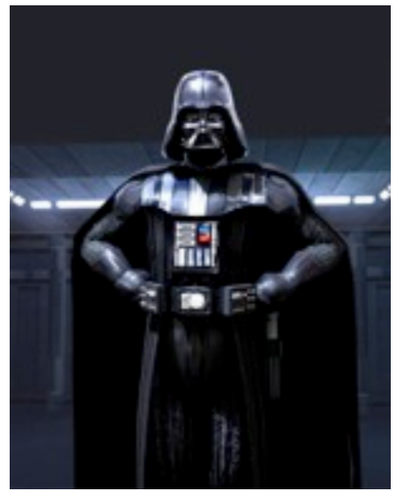
- Cherchent à obtenir de l'information supplémentaire sur le fonctionnement de la boîte noire:
  - Timing
  - Emanations diverses
  - Fautes

Adversaires de type «grey-box»

- Cherchent à obtenir de l'information supplémentaire sur le fonctionnement de la boîte noire:
  - Timing
  - Emanations diverses
  - Fautes

Adversaires de type «white-box»

- C’est le type d’adversaire contre lequel il est le plus difficile de se protéger.



- Il maîtrise complètement le SW/HW
- Il peut lire dans toutes les mémoires
- Il peut perturber tous les calculs

Adversaire de type «white-box»

- Exemples dans la vie réelle:
  - Contournement de DRM
  - «Cracking» des mécanismes de licence
  - Vol de propriété intellectuelle par «reverse-engineering» de SW

Adversaires de type «white-box»

- Exemple classique:

```
if (RSA_verify (signature) ==
RSA_VALID_SIGNATURE) {

  // Perform some critical operation
  } else {
    return NOT_AUTHENTICATED
  }
```

Adversaires de type «white-box»

- Au niveau du langage machine, cela donne:

Adversaires de type «white-box»

- Au niveau du langage machine, cela donne:

```
...
cmp     $0x0, %ebx
je      0x64FE89A1
...
```

Adversaires de type «white-box»

- Au niveau du langage machine, cela donne:

```
...
cmp     $0x0, %ebx
je      0x64FE89A1
...
```

La sécurité de la signature RSA repose sur le fait que cette instruction soit exécutée ou pas...

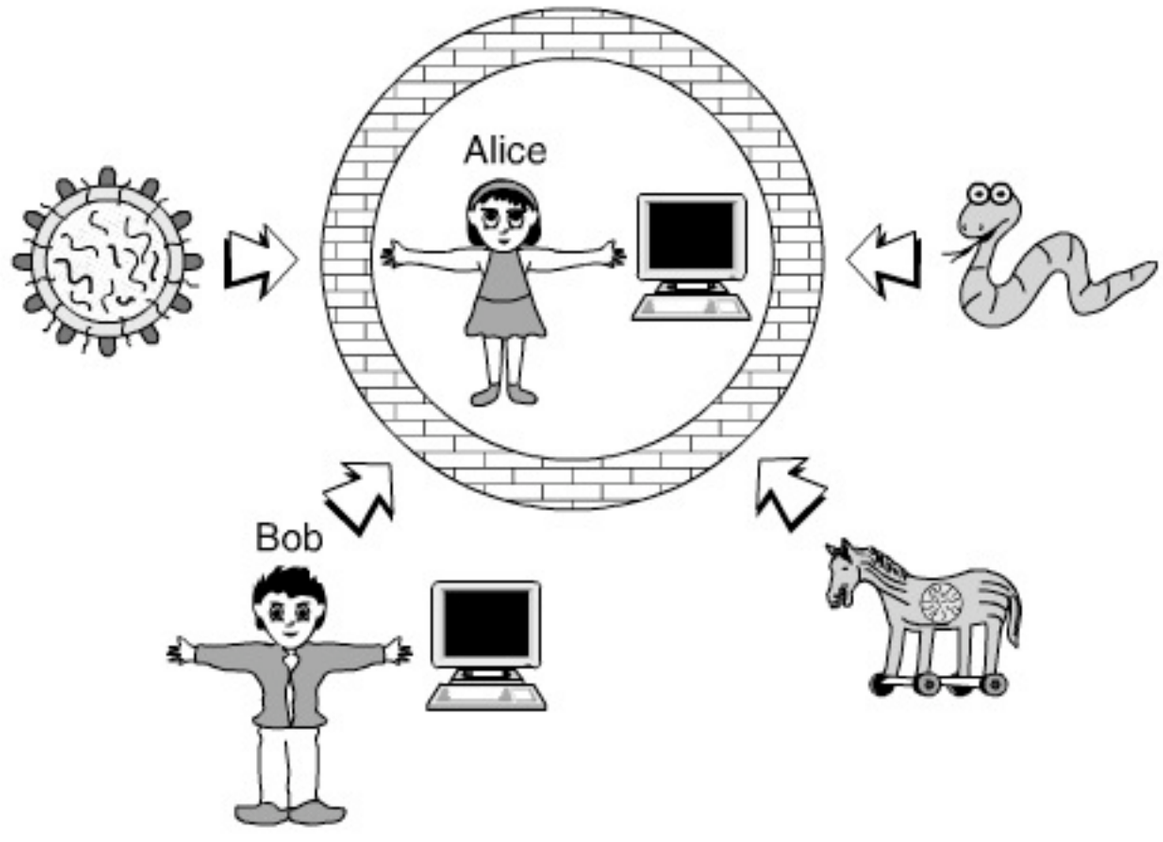
Contre-Mesures Classiques

- Utilisation de «hardware» comme base de confiance:
  - TPM
  - smartcard
  - USB dongle
  - …

Contre-Mesures Classiques

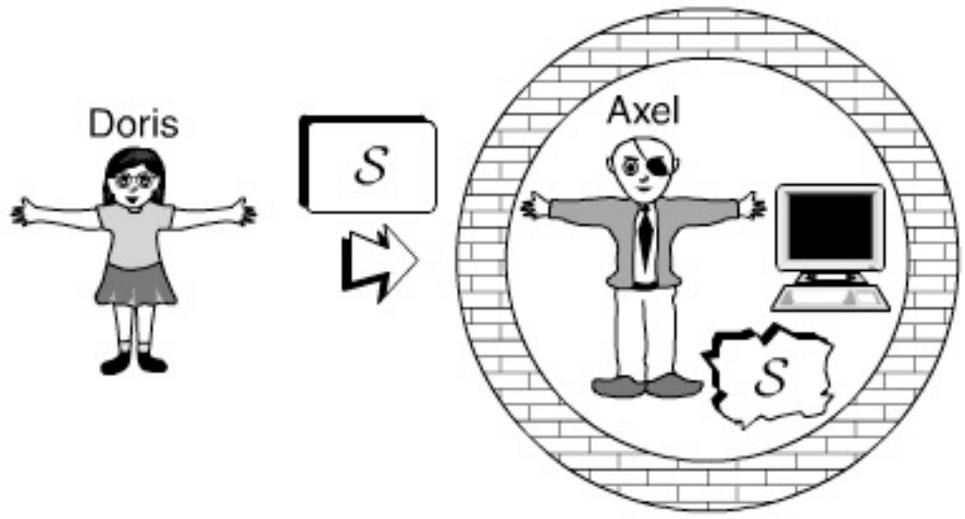
- Problème: ce hardware n'est pas toujours:
  - présent;
  - suffisamment bon marché;
  - flexible;
  - etc.

Protection Logicielle



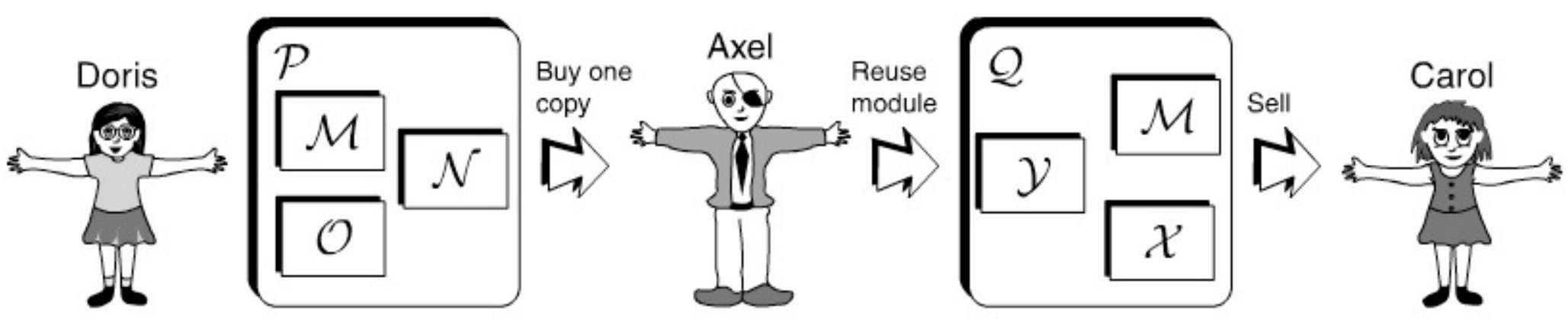
Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009

Protection Logicielle

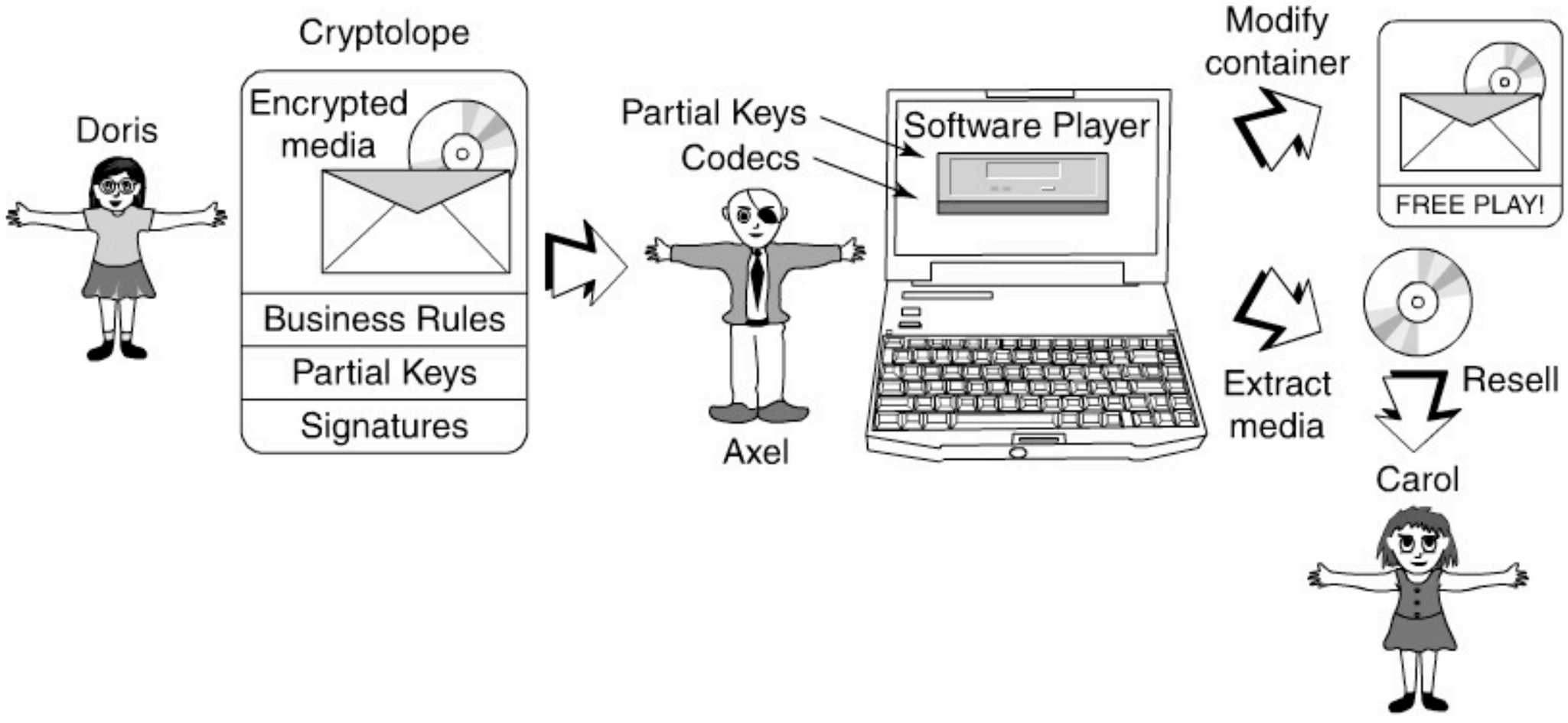


Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009

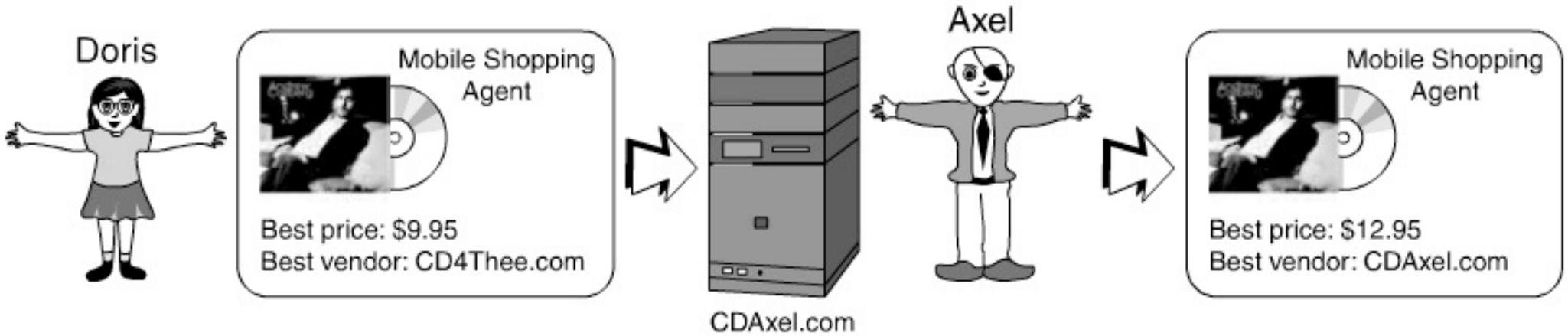
«Reverse Engineering»



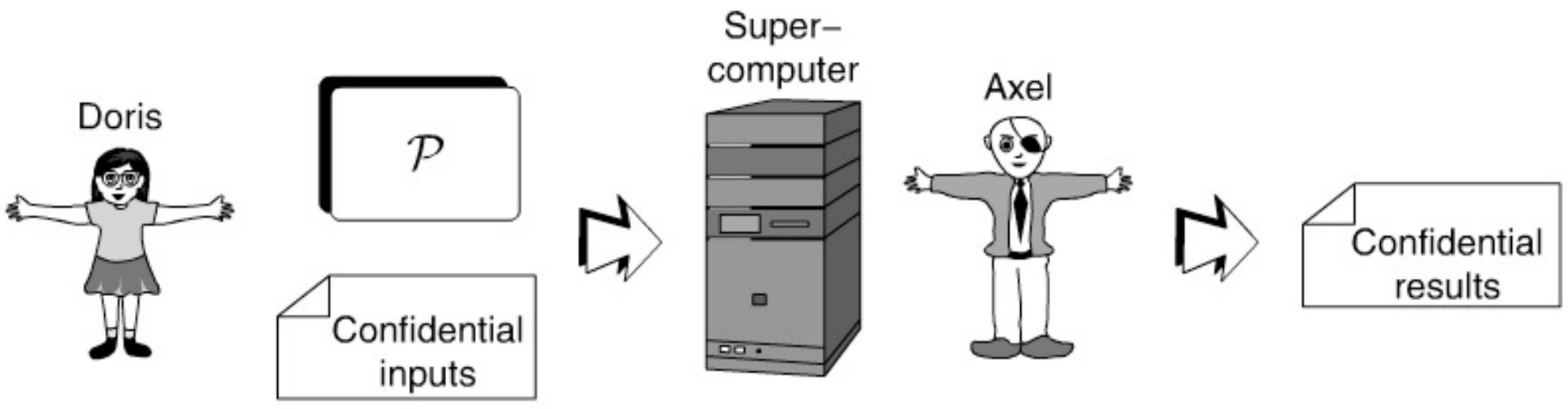
DRM



Agents Mobiles



«Cloud Computing»



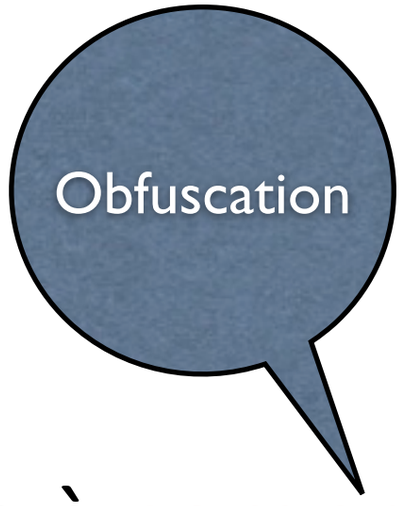
Scénarios d'attaque

- «Reverse Engineering», extraction d'IP, de secrets
- Modification de code
- Distribution de code

Protection Logicielle

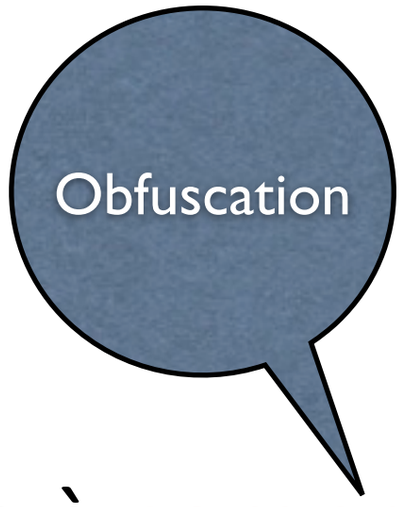
- Rendre le code plus difficile à comprendre
- Rendre le code plus difficile à modifier
- Rendre le code unique

Protection Logicielle

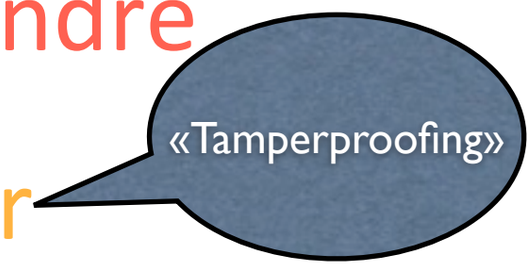


- Rendre le code plus difficile à comprendre
- Rendre le code plus difficile à modifier
- Rendre le code unique

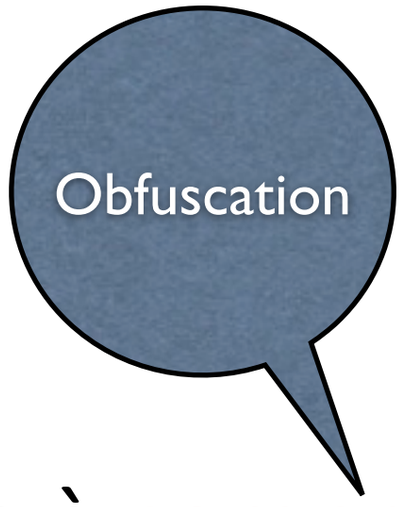
Protection Logicielle



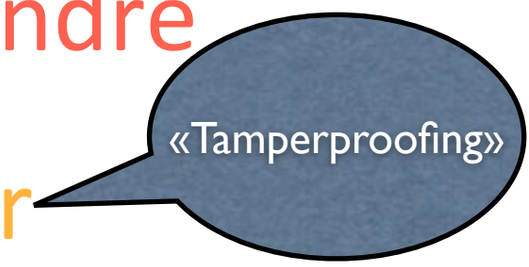
- Rendre le code plus difficile à comprendre
- Rendre le code plus difficile à modifier
- Rendre le code unique

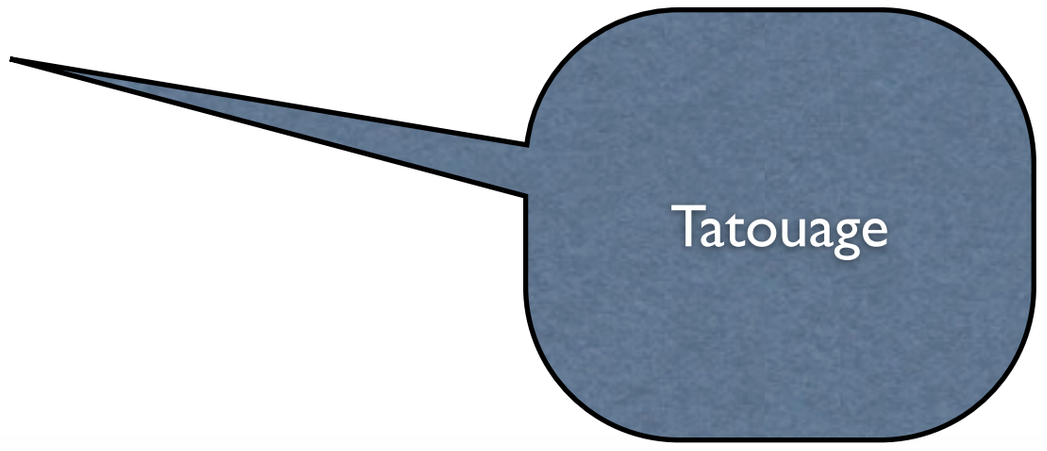


Protection Logicielle



- Rendre le code plus difficile à comprendre
- Rendre le code plus difficile à modifier
- Rendre le code unique





Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Techniques d'obfuscation

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Software Protection

- D’après Wikipedia, un logiciel «obfusqué» est du code source ou machine qui a été rendu difficile à comprendre.

Protection Logicielle

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich¶
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an *obfuscator* $\mathcal{O}$ is an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is "unintelligible" in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice's theorem. Most of these applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that $\mathcal{O}(P)$ is a "virtual black box," in the sense that anything one can efficiently compute given $\mathcal{O}(P)$, one could also efficiently compute given oracle access to P.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $\mathcal{P}$ that are *unobfuscatable* in the sense that (a) given *any* efficient program P' that computes the same function as a program $P \in \mathcal{P}$, the "source code" P can be efficiently reconstructed, yet (b) given *oracle access* to a (randomly selected) program $P \in \mathcal{P}$, no efficient algorithm can reconstruct P (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\mathbf{TC^0}$). We also rule out several potential applications of obfuscators, by constructing "unobfuscatable" signature schemes, encryption schemes, and pseudorandom function families.

Protection Logicielle

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich¶
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an *obfuscator* $\mathcal{O}$ is an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is "unintelligible" in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice's theorem. Most of these applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that $\mathcal{O}(P)$ is a "virtual black box," in the sense that anything one can efficiently compute given $\mathcal{O}(P)$, one could also efficiently compute given oracle access to P.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $\mathcal{P}$ that are *unobfuscatable* in the sense that (a) given *any* efficient program P' that computes the same function as a program $P \in \mathcal{P}$, the "source code" P can be efficiently reconstructed, yet (b) given *oracle access* to a (randomly selected) program $P \in \mathcal{P}$, no efficient algorithm can reconstruct P (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\mathbf{TC^0}$). We also rule out several potential applications of obfuscators, by constructing "unobfuscatable" signature schemes, encryption schemes, and pseudorandom function families.

Protection Logicielle

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich¶
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an *obfuscator* $\mathcal{O}$ is an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is "unintelligible" in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice's theorem. Most of these applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that $\mathcal{O}(P)$ is a "virtual black box," in the sense that anything one can efficiently compute given $\mathcal{O}(P)$, one could also efficiently compute given oracle access to P.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $\mathcal{P}$ that are *unobfuscatable* in the sense that (a) given *any* efficient program P' that computes the same function as a program $P \in \mathcal{P}$, the "source code" P can be efficiently reconstructed, yet (b) given *oracle access* to a (randomly selected) program $P \in \mathcal{P}$, no efficient algorithm can reconstruct P (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\mathbf{TC}^0$). We also rule out several potential applications of obfuscators, by constructing "unobfuscatable" signature schemes, encryption schemes, and pseudorandom function families.

Protection Logicielle

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich¶
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an *obfuscator* $\mathcal{O}$ is an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is "unintelligible" in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice's theorem. Most of these applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that $\mathcal{O}(P)$ is a "virtual black box," in the sense that anything one can efficiently compute given $\mathcal{O}(P)$, one could also efficiently compute given oracle access to P.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $\mathcal{P}$ that are *unobfuscatable* in the sense that (a) given *any* efficient program P' that computes the same function as a program $P \in \mathcal{P}$, the "source code" P can be efficiently reconstructed, yet (b) given *oracle access* to a (randomly selected) program $P \in \mathcal{P}$, no efficient algorithm can reconstruct P (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\mathbf{TC}^0$). We also rule out several potential applications of obfuscators, by constructing "unobfuscatable" signature schemes, encryption schemes, and pseudorandom function families.

Protection Logicielle

On the (Im)possibility of Obfuscating Programs*

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich¶
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an *obfuscator* $\mathcal{O}$ is an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is "unintelligible" in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice's theorem. Most of these applications are based on an interpretation of the "unintelligibility" condition in obfuscation as meaning that $\mathcal{O}(P)$ is a "virtual black box," in the sense that anything one can efficiently compute given $\mathcal{O}(P)$, one could also efficiently compute given oracle access to P.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $\mathcal{P}$ that are *unobfuscatable* in the sense that (a) given *any* efficient program P' that computes the same function as a program $P \in \mathcal{P}$, the "source code" P can be efficiently reconstructed, yet (b) given *oracle access* to a (randomly selected) program $P \in \mathcal{P}$, no efficient algorithm can reconstruct P (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($\mathbf{TC}^0$). We also rule out several potential applications of obfuscators, by constructing "unobfuscatable" signature schemes, encryption schemes, and pseudorandom function families.

Protection Logicielle

- Même si c'est impossible en théorie, rien ne nous empêche d'essayer en pratique :-)
- Hypothèses de sécurité plus faibles ?
  - Le but est de rendre la tâche à l'adversaire plus coûteuse, même si on n'obtient pas une résistance de type cryptographique...

Protection Logicielle

```
return (int) ((((x - 2) * (x - 3) * (x - 4) * (x - 5) * (x - 6) *
                 (x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
                 (x - 12) * 31) /
                ((x - 2) * (x - 3) * (x - 4) * (x - 5) * (x - 6) *
                 (x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
                 (x - 12) + .00001)) +
               (((x - 3) * (x - 4) * (x - 5) * (x - 6) * (x - 7) *
                 (x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) *
                 (28 + z)) /
                ((x - 3) * (x - 4) * (x - 5) * (x - 6) * (x - 7) *
                 (x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) +
                 .00001)) +
               (((x - 4) * (x - 5) * (x - 6) * (x - 7) * (x - 8) *
                 (x - 9) * (x - 10) * (x - 11) * (x - 12) * 31) /
                ((x - 4) * (x - 5) * (x - 6) * (x - 7) * (x - 8) *
                 (x - 9) * (x - 10) * (x - 11) * (x - 12) + .00001)) +
               (((x - 5) * (x - 6) * (x - 7) * (x - 8) * (x - 9) *
                 (x - 10) * (x - 11) * (x - 12) * 30) /
                ((x - 5) * (x - 6) * (x - 7) * (x - 8) * (x - 9) *
                 (x - 10) * (x - 11) * (x - 12) + .00001)) +
               (((x - 6) * (x - 7) * (x - 8) * (x - 9) * (x - 10) *
                 (x - 11) * (x - 12) * 31) /
                ((x - 6) * (x - 7) * (x - 8) * (x - 9) * (x - 10) *
                 (x - 11) * (x - 12
                 ) + .00001)) +
               (((x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
                 (x - 12) * 30) /
                ((x - 7) * (x - 8) * (x - 9) * (x - 10) * (x - 11) *
                 (x - 12) + .00001)) +
               (((x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) *
                 31) /
                ((x - 8) * (x - 9) * (x - 10) * (x - 11) * (x - 12) +
                 .00001)) +
               (((x - 9) * (x - 10) * (x - 11) * (x - 12) * 31) /
                ((x - 9) * (x - 10) * (x - 11) * (x - 12) + .00001)) +
               (((x - 10) * (x - 11) * (x - 12) * 30) /
                ((x - 10) * (x - 11) * (x - 12) + .00001)) +
               (((x - 11) * (x - 12) * 31) /
                ((x - 11) * (x - 12) + .00001)) +
               (((x - 12) * 30) / ((x - 12) + .00001)) + 31 + .1) -
y;
```

Source: Wikipedia

Protection Logicielle

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU /
lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=
$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}
p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=
$d[$q];sleep rand(2)if/\S/;print
```

Source: Wikipedia

Protection Logicielle

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU /
lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=
$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}
p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=
$d[$q];sleep rand(2)if/\S/;print
```

Ce petit programme en Perl affiche le texte «Just another Perl/Unix hacker», plusieurs caractères à la fois, avec des délais.

Source: Wikipedia

Protection Logicielle

```
void primes(int cap) {
  int i, j, composite;
  for(i = 2; i < cap; ++i) {
    composite = 0;
    for(j = 2; j * j <= i; ++j)
      composite += !(i % j);
    if(!composite)
      printf("%d\t", i);
  }
}

int main(void) {
  primes(100);
}
```

Source: Wikipedia

Protection Logicielle

```
void primes(int cap) {
  int i, j, composite;
  for(i = 2; i < cap; ++i) {
    composite = 0;
    for(j = 2; j * j <= i; ++j)
      composite += !(i % j);
    if(!composite)
      printf("%d\t", i);
  }
}

int main(void) {
  primes(100);
}
```

Un crible d'Erastothène

Source: Wikipedia

Protection Logicielle

```
void primes(int cap) {
  int i, j, composite;
  for(i = 2; i < cap; ++i) {
    composite = 0;
    for(j = 2; j * j <= i; ++j)
      composite += !(i % j);
    if(!composite)
      printf("%d\t", i);
  }
}

int main(void) {
  primes(100);
}
```

Un crible d'Erastothène

Source: Wikipedia

Protection Logicielle

```
void primes(int cap) {
  int i, j, composite;
  for(i = 2; i < cap; ++i) {
    composite = 0;
    for(j = 2; j * j <= i; ++j)
      composite += !(i % j);
    if(!composite)
      printf("%d\t", i);
  }
}

int main(void) {
  primes(100);
}
```

```
_(__,___,____,_____){___/__<=_____?_(__,___
+_____,____,_____):!(___%__)?_(__,___
+_____,___%__,_____):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___
+_____,____,_____)):(___%__>_____&&___
%__<___/__)?_(__,___+
_____,____,_____+!(___/__%(___
%__))):___<__*__?_(__,___
+_____,____,_____):0;}main(void)
{_(100,0,0,1);}
```

Un crible d'Erastothène

Source: Wikipedia

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Protection Logicielle

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Protection Logicielle

Protection Logicielle

- Scénarios différents
  - Code source vs. binaire
  - Langages supportés
    - .NET, C#, Java, Javascript, C/C++, (Fortran, Ada, Haskell, Python ?)
- Coûts
  - Taille et vitesse du code
- Résistance au «reverse engineering»

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Techniques avancées de protection logicielle

Packing

Les «packers», autrefois utilisés pour compresser un exécutable, sont également utilisés à des fins de protection logicielle.

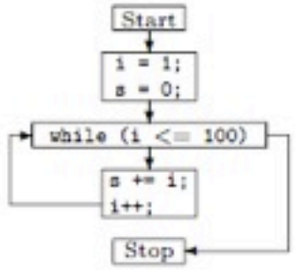
Portable Executable

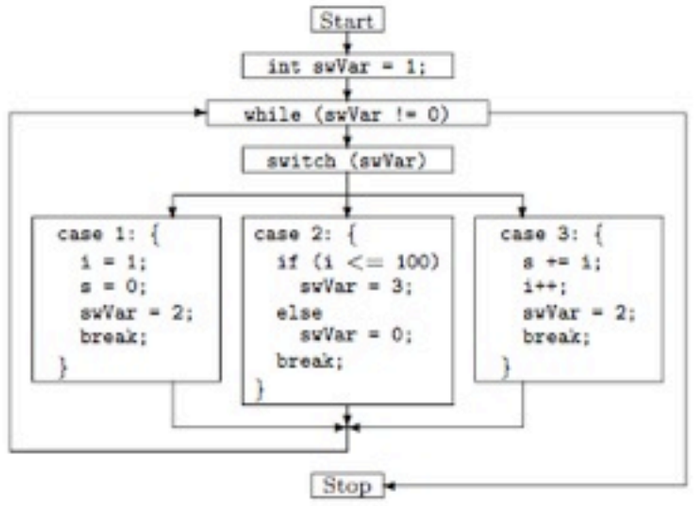
- ASPack
- ASPR (ASProtect)
- Armadillo Packer
- AxProtector
- BeRoEXEPacker
- CExe
- exe32pack
- EXE Bundle
- EXECryptor
- EXE Stealth
- eXPressor
- Enigma Protector Win32 / Win64
- Enigma Virtual BOX Freeware
- MPRESS – Freeware
- FSG (Fast Small Good)
- HASP Envelope
- kkrunchy – Freeware
- MEW – development stopped
- Npack - Freeware
- NeoLite
- Obsidium
- PECompact
- PEPack
- PKLite32
- PELock
- PESpin
- PEtite
- Privilege Shell
- RLPack
- Sentinel CodeCover (Sentinel Shell)
- Shrinker32
- Smart Packer Pro
- SmartKey GSS
- tElock
- Themida
- UniKey Enveloper
- Upack (software) – Freeware
- UPX – free software
- VMProtect
- WWPack
- BoxedApp Packer
- XComp/XPack – Freeware

«Code Flattening»

```
                                int swVar = 1;
                                while (swVar != 0) {
                                  switch (swVar) {
                                    case 1: {
i = 1;                                i = 1;
s = 0;                                s = 0;
                                      swVar = 2;
                                      break;
                                    }
                                    case 2: {
while (i <= 100) {                    if (i <= 100)
                                        swVar = 3;
                                      else
                                        swVar = 0;
                                      break;
                                    }
                                    case 3: {
  s += i;                             s += i;
  i++;                                i++;
                                      swVar = 2;
                                      break;
                                    }
                                  }
}                               }
```

(a) (b)





Prédicats Opaques

- Un prédicat opaque est un prédicat booléen tel que:
  - Le programmeur en connaît la valeur;
  - Le «reverse-engineer» est forcé d'en étudier l'exécution pour se rendre compte que le code obtient toujours le même résultat
- Force à passer dans le domaine de l'analyse dynamique («debugging», émulation, ...)

Table 1: Examples of number-theoretical true opaque predicates

$\forall x, y \in \mathbf{Z}$	:	$7y^2 - 1 \neq x^2$
$\forall x \in \mathbf{Z}$	:	$3 \mid (x^3 - x)$
$\forall x \in \mathbf{N}$	:	$14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$
$\forall x \in \mathbf{Z}$	:	$2 \mid x \vee 8 \mid (x^2 - 1)$
$\forall x \in \mathbf{Z}$	:	$\sum_{i=1,\, 2 \nmid i}^{2x-1} i = x^2$
$\forall x \in \mathbf{N}$	:	$2 \mid \lfloor \frac{x^2}{2} \rfloor$

«Code Interleaving»

- L'idée consiste à «mélanger» des portions de code indépendants (donc parallélisables)
- On peut y ajouter du «junk code» à volonté
- Fusion de procédures
- «Re-splitting» dans des threads différents
- Etc.

Virtualisation

- Traduire le software en un «byte-code» individualisé, et l'exécuter dans une machine virtuelle «custom»
- Le concept peut être itéré, mais attention aux performances...
- Utiliser des architectures Turing-complètes exotiques
  - `brainfuck` (8 instructions, pas d'opérandes)
  - `subleq` (1 instruction, 3 opérandes)
  - ...

Ajouter une dimension temporelle

- Si un réseau est disponible, on peut exploiter la dimension temporelle:
  - À des moments aléatoires, le serveur demande au client une somme de contrôle sur une partie aléatoire du code
  - Délai de réponse: moins d'une seconde.
  - Permet de détecter des modifications du logiciel
- Largement utilisé par l'industrie du jeu en ligne pour éviter la triche

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Obfuscation avec LLVM

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Motivations

- Constation de départ: il n'existe pas d'outil open-source «industrialisable» capable d'obfusquer du C/C++ de façon satisfaisante.

- «Reverse engineering is hard, protection against RE is even harder, so let's face the challenge» !

Approches abandonnées

- Parser le code source, le transformer, et re-générer du code source

SPLAT - A Simple Python Library for Abstract syntax tree Transformation

Sébastien Bischof
Professor: Dr. Pascal Junod

June 17, 2010

- Outil en Python écrit par Sébastien Bischof, se basant sur un parser existant, et implémentant du «code flattening»

Approches abandonnées

- Désavantages majeurs de cette approche:
  - Difficulté d'écrire un parser complet et robuste
  - Un parser ne supporte qu'un seul langage: le travail doit être recommencé pour chaque nouveau langage supporté

Approches abandonnées

- Utiliser un «front-end» existant, et le modifier pour y ajouter des techniques d'obfusquation

- PA de Grégory Ruch, basé sur les API de Clang, le «front-end» C/ C++ de LLVM

Obfuscator - Abstract syntax tree Transformation

Grégory Ruch
Professeur : Dr. Pascal Junod

29 avril 2011

Approches abandonnées

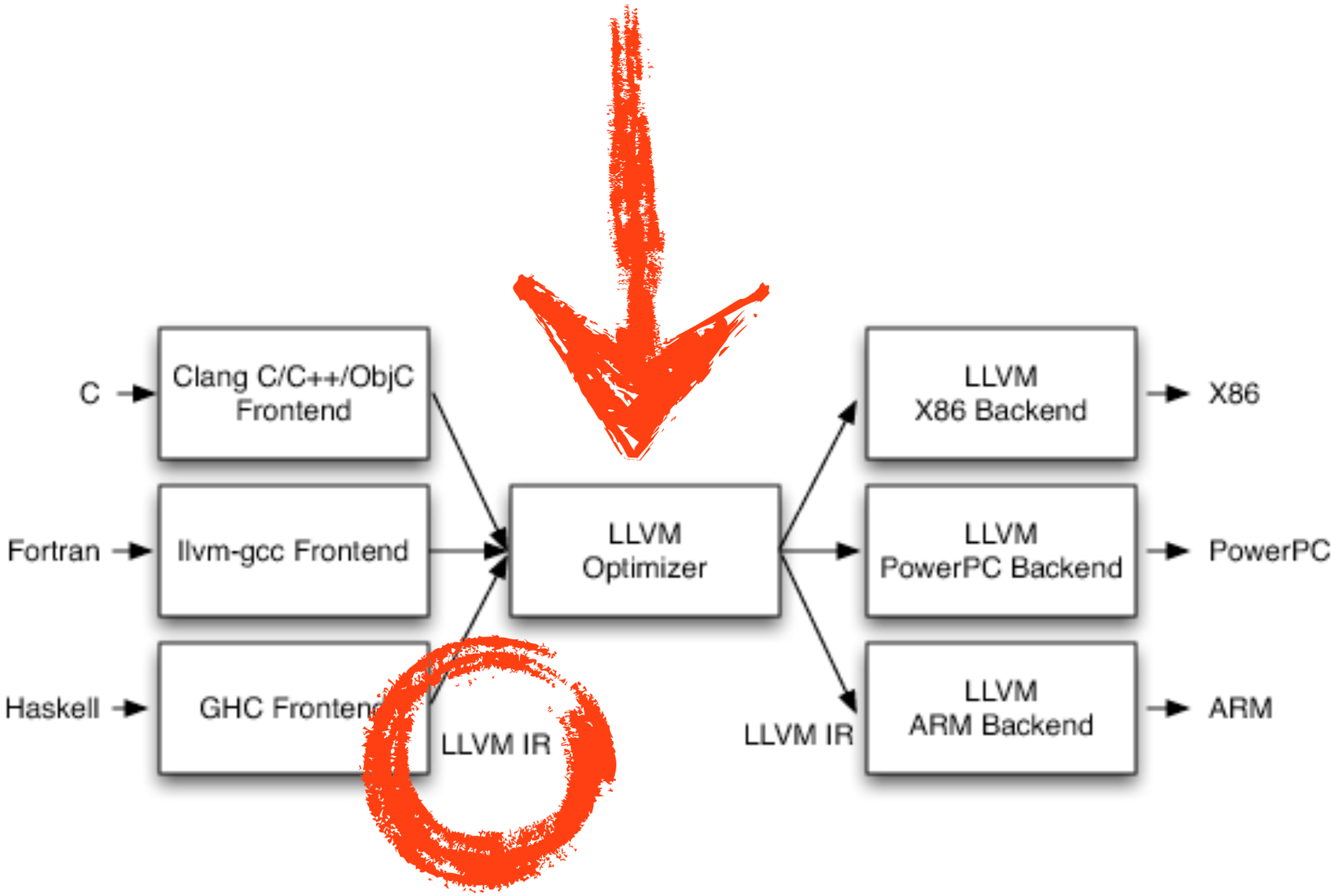
- Désavantages majeurs de cette approche:
  - Les APIs de Clang n'ont pas été conçues pour effectuer des modifications de l'AST généré.
  - À nouveau, l'outil résultant est restreint à quelques langages.

LLVM

- Infrastructure complète de compilation
- Projet open-source commencé à l'université de l'Illinois en 2000
- Depuis 2005, le principal sponsor est Apple Inc., qui a engagé Chris Lattner
- Communauté très dynamique
- Architecture logicielle «state-of-the-art»

LLVM

- Front-ends:
  - C, C++, Objective C, Fortran, Ada, Haskell, Python, Ruby, …
- Back-ends:
  - x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, Sparc, Alpha, MIPS, MSP430, SystemZ, XCore


C
Clang C/C++/ObjC
Frontend
Fortran
llvm-gcc Frontend
Haskell
GHC Frontend
LLVM IR
LLVM
Optimizer
LLVM IR
LLVM
X86 Backend
X86
LLVM
PowerPC Backend
PowerPC
LLVM
ARM Backend
ARM

Listing 3.1 – Simple fonction faisant une addition en C

```
1 int addition(int a, int b){
2     return a + b;
3 }
```

Listing 3.2 – Simple fonction faisant une addition en LLVM-IR

```
1 define i32 @addition(i32 %a, i32 %b) nounwind readnone {
2 entry:
3   %1 = add i32 %a, %b
4   ret i32 %1
5 }
```

LLVM et obfuscation

- LLVM offre une API très complète et documentée pour «jouer» avec l'IR
- L'idée (entre autres) est de pouvoir écrire des passes d'optimisation de manière efficace
- Notre idée: au lieu d'écrire un passe d'optimisation, on va faire de l'obfuscation!

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

LLVM et obfuscation

```
$ clang -O3 -o prog prog.c
```

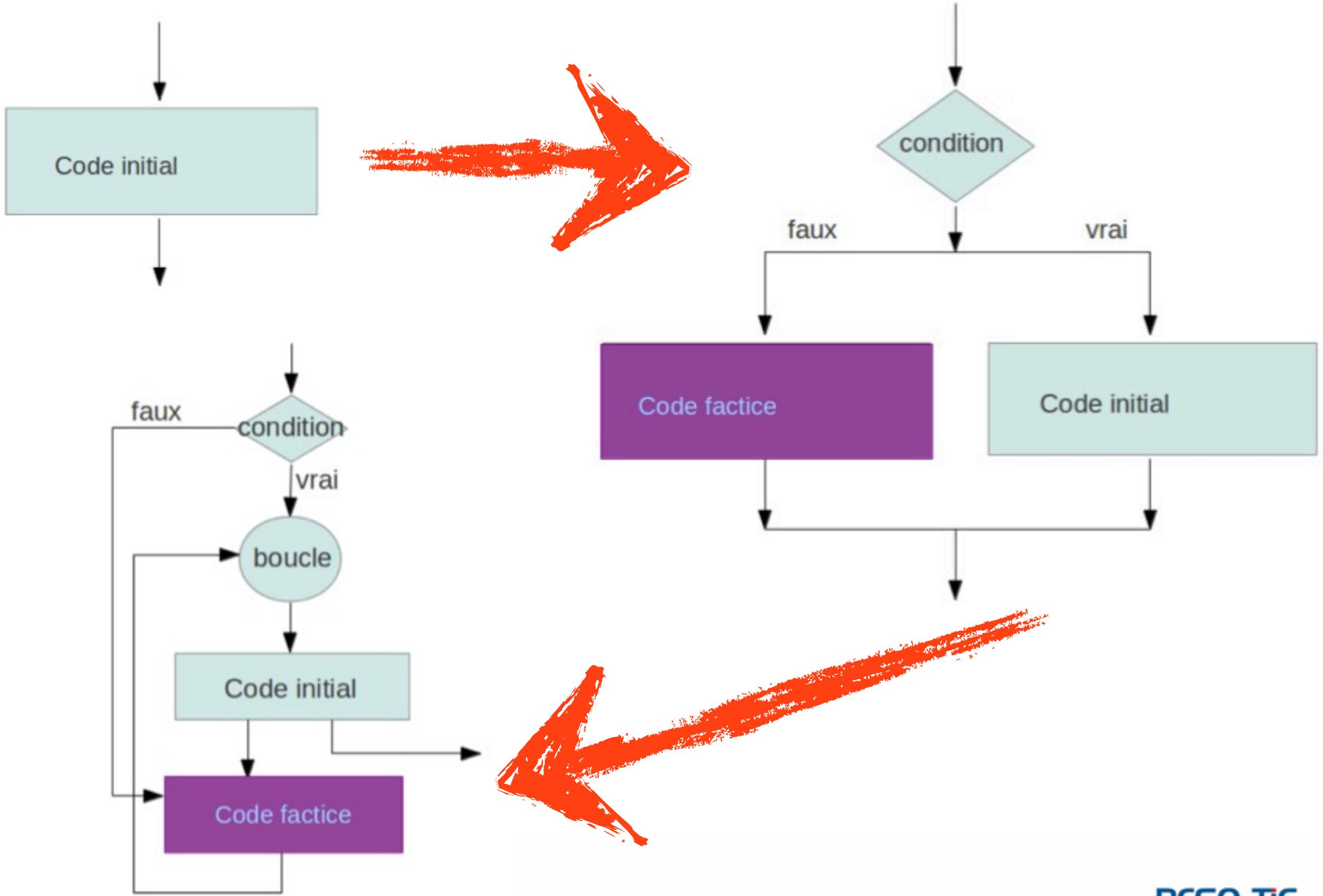
```
$ clang -O3 -B3 -o prog prog.c
```

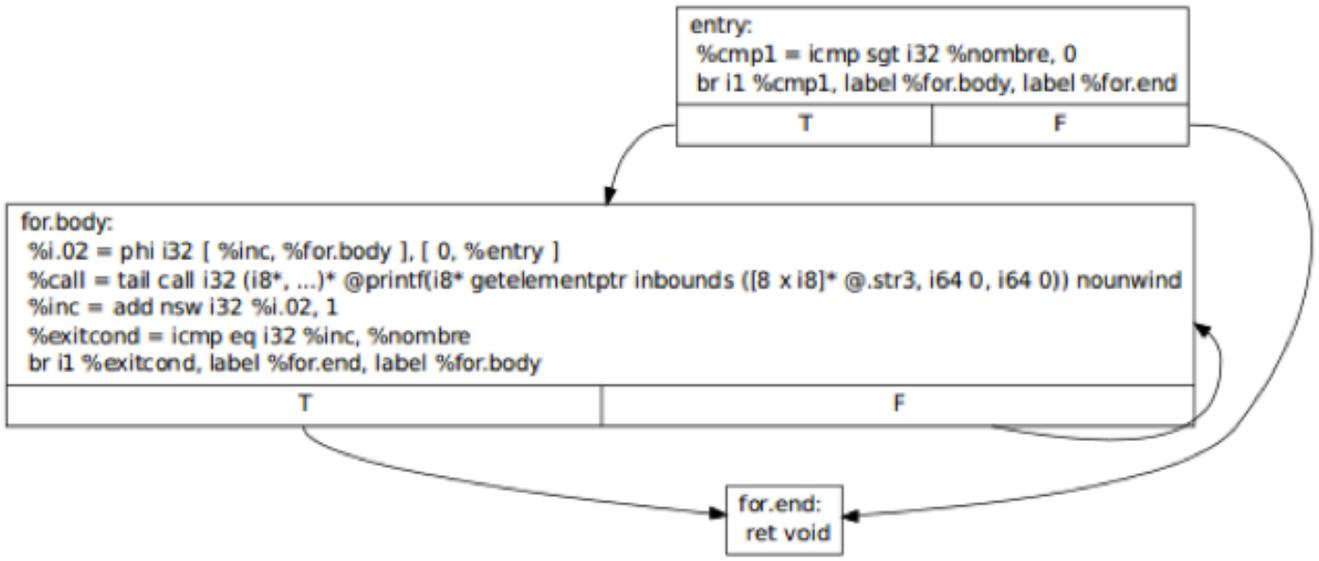
Substitution de Code

- Première passe écrite, «pour se faire la main»
- Remplacer une instruction de base par une autre représentation:
  - `A ^ B = (A & ~B) | (~A & B)`
  - `A + B = A - (-B)`
  - `A+B = (A+R) + (B+R) - 2*R`
  - `...`

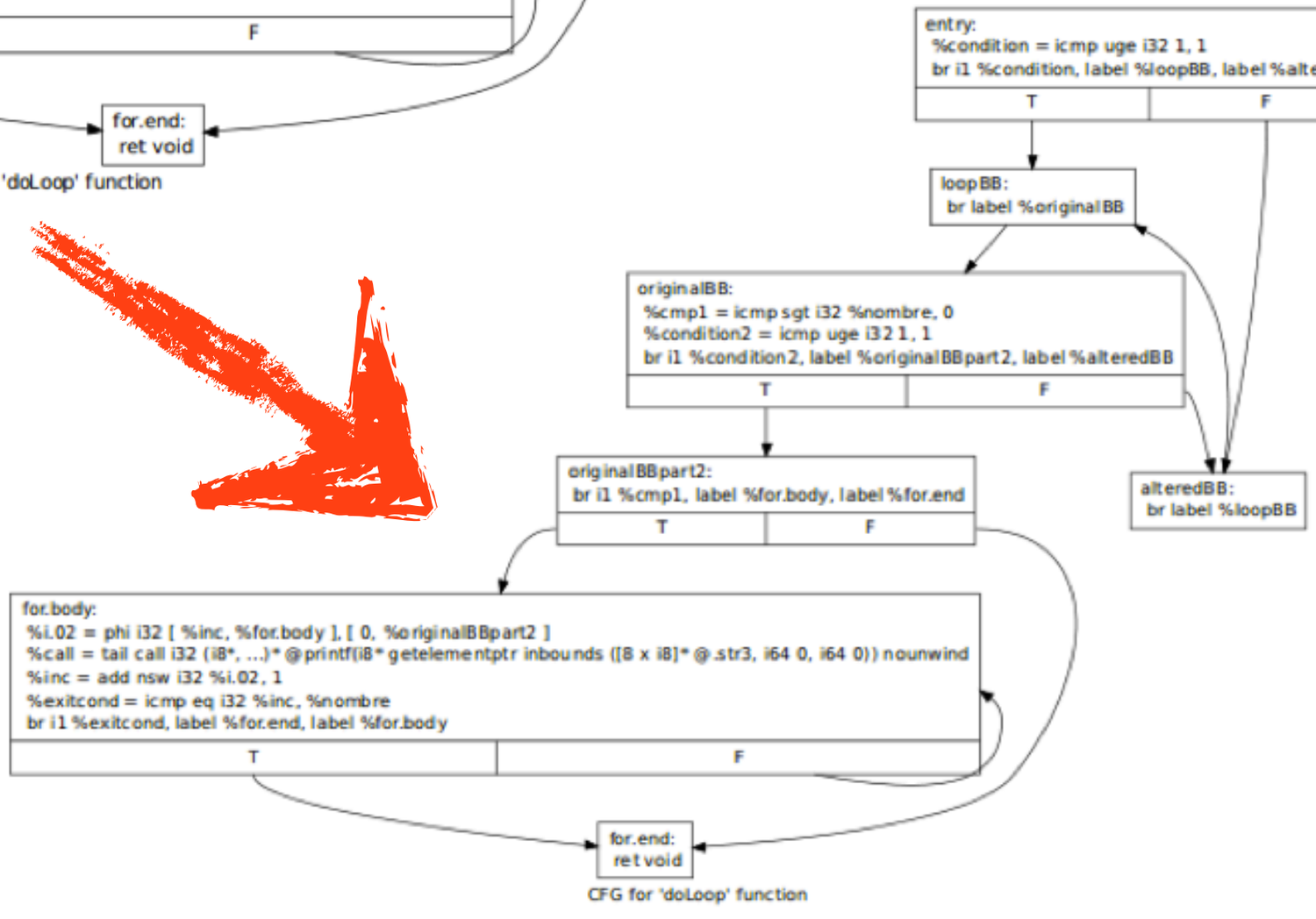
Insertion de faux branchements

- Thèse de bachelor de Julie Michielin
- Idées de base:
  - Insérer de faux branchements
  - Rendre le graphe de flux irréductible
  - Utiliser un prédicat opaque


Code initial
condition
faux
vrai
Code factice
Code initial
faux
condition
vrai
boucle
Code initial
Code factice



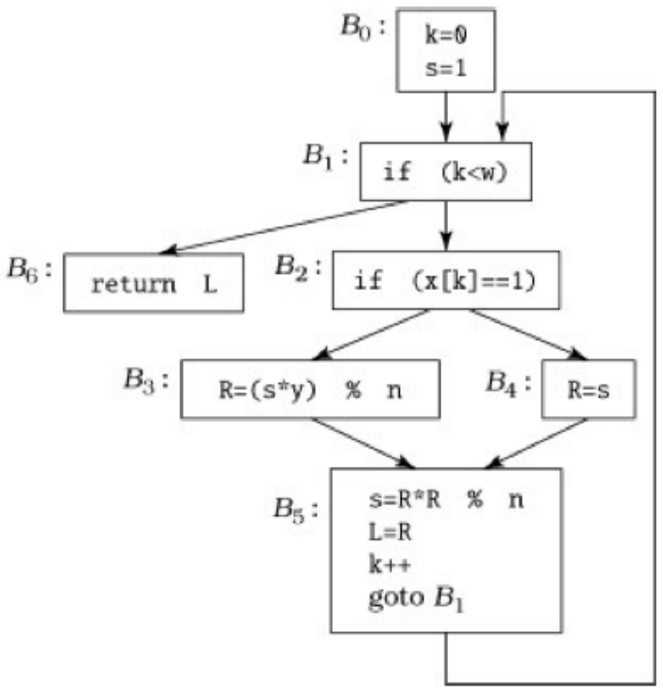
CFG for 'doLoop' function

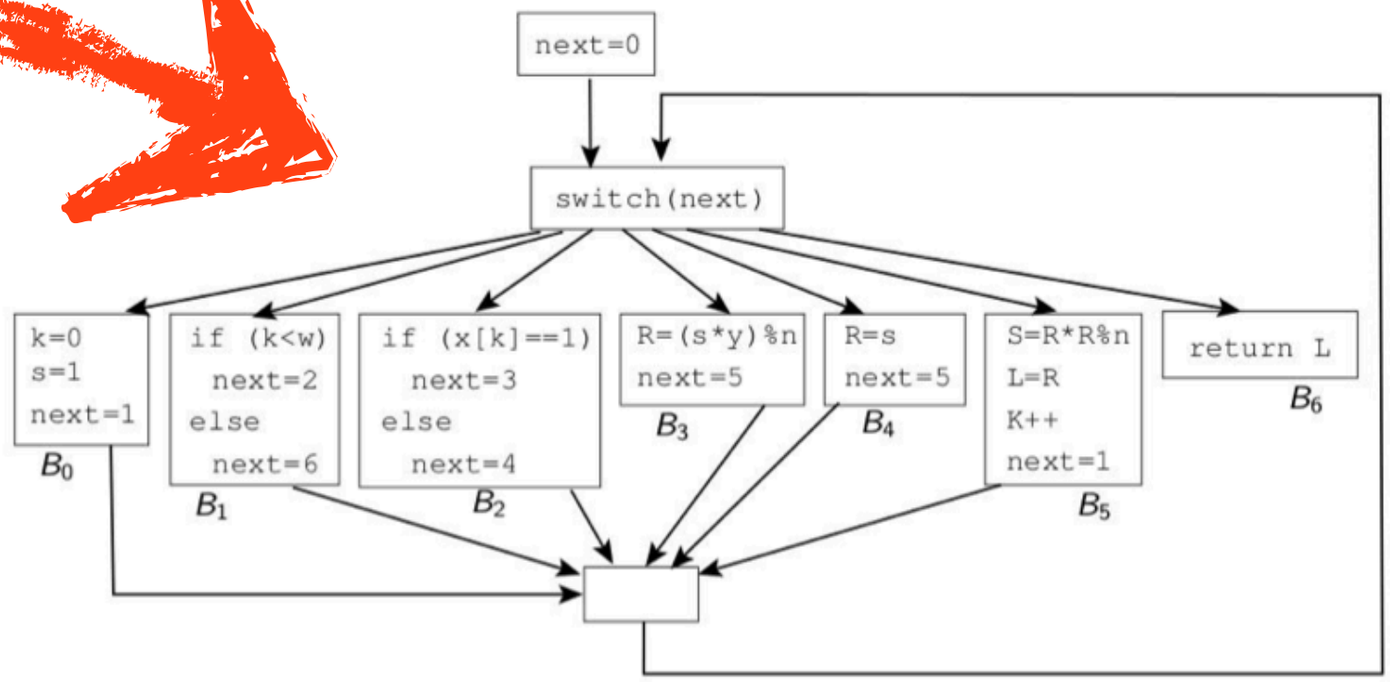


CFG for 'doLoop' function

«Code Flattening»

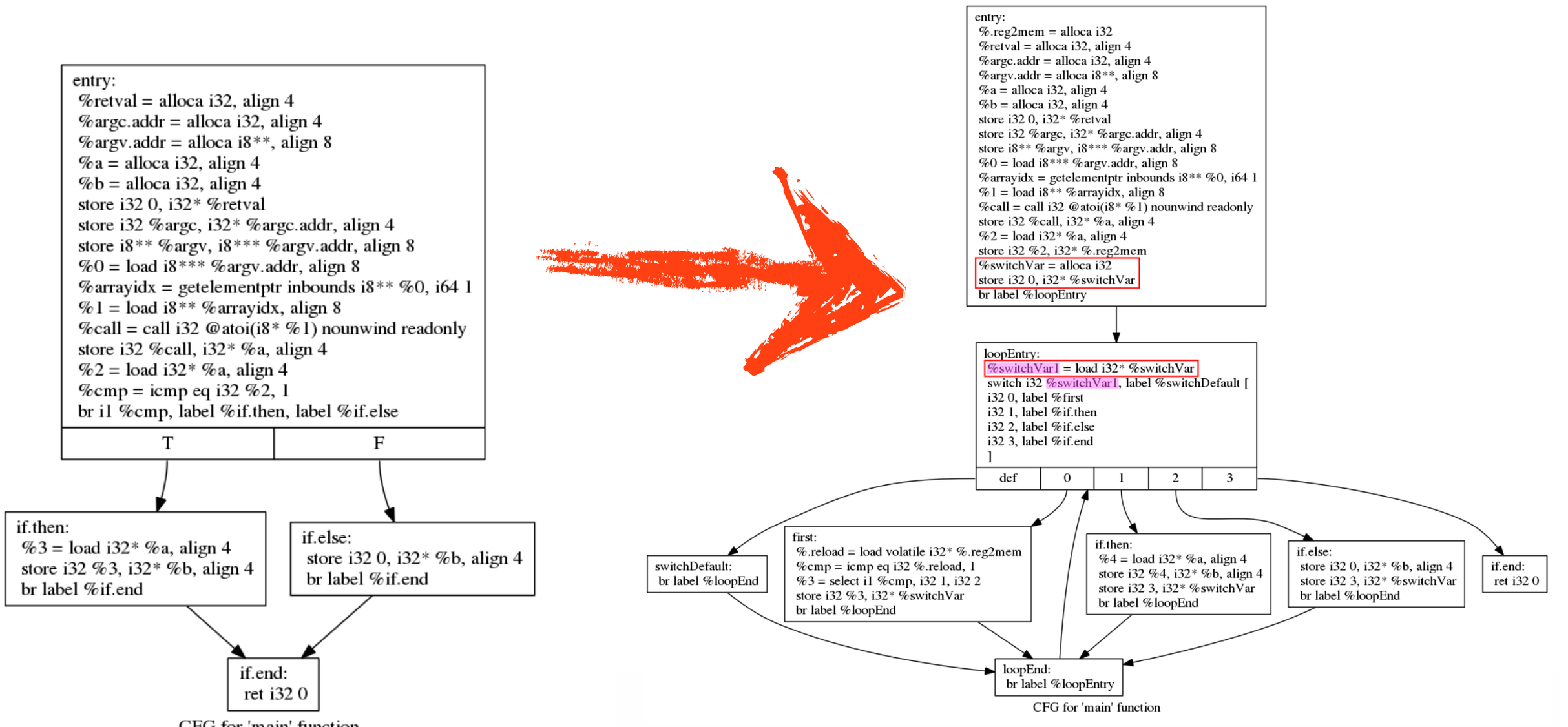
- De loin la passe la plus complexe et la plus difficile à mettre en oeuvre





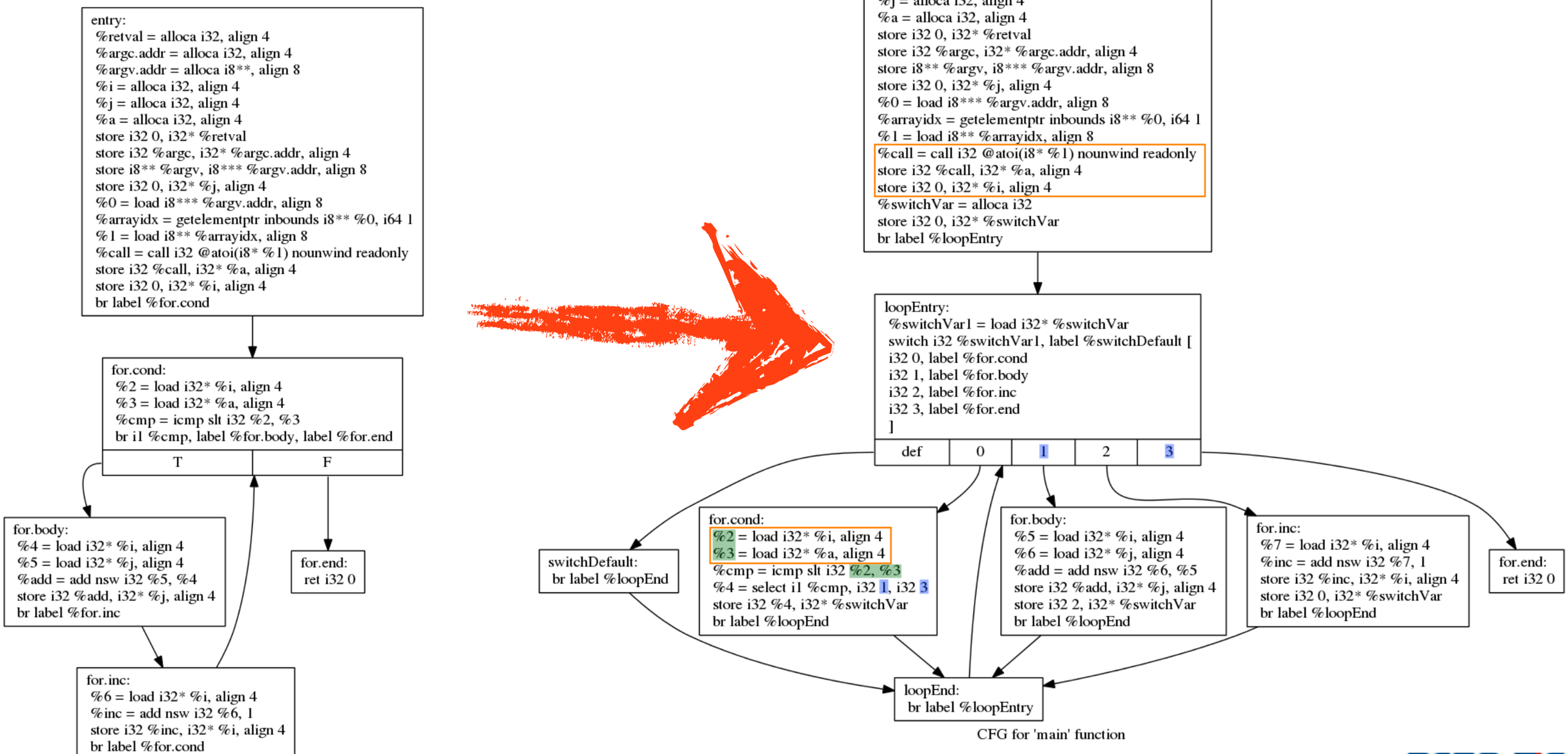
«Code Flattening»

- Cas d'un IF-THEN-ELSE:



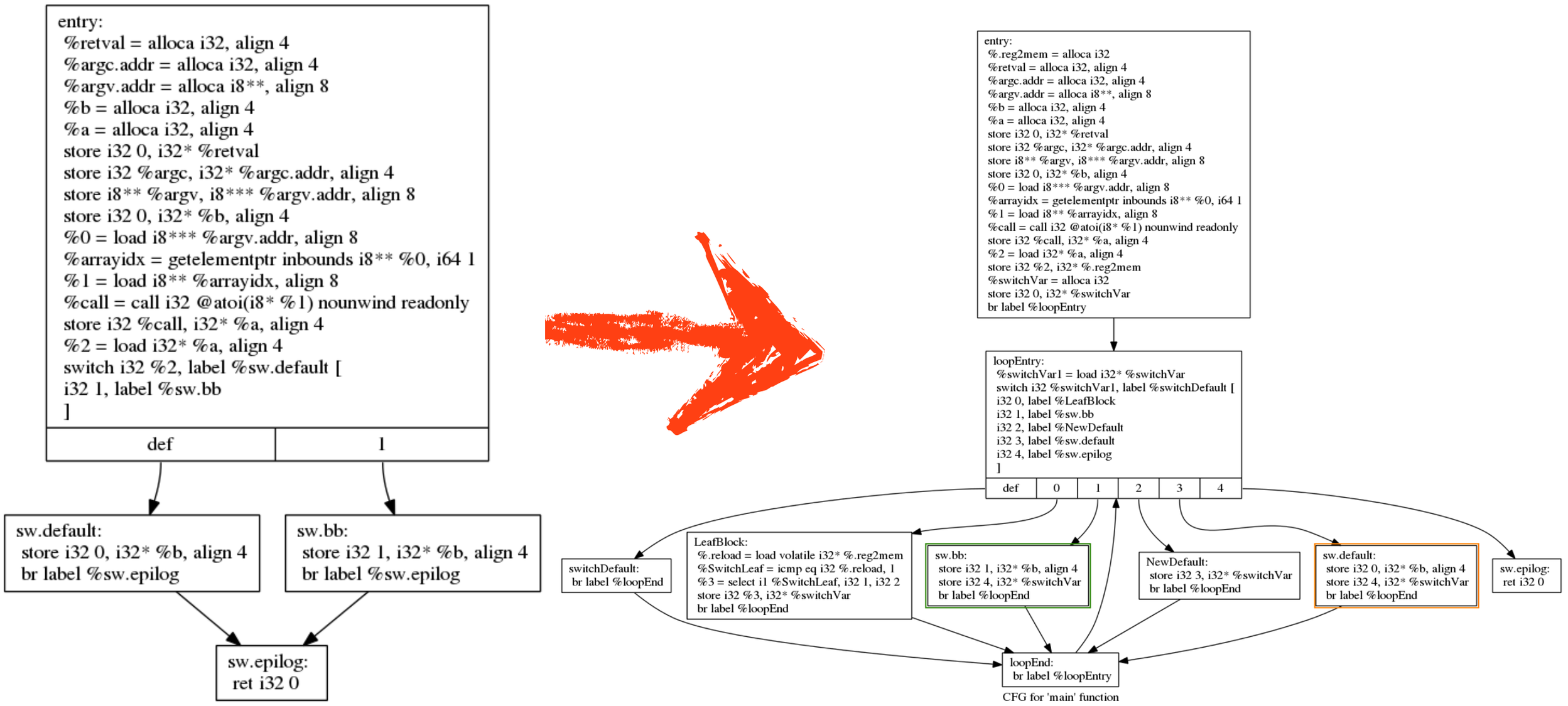
«Code Flattening»

- Cas d'une boucle FOR:



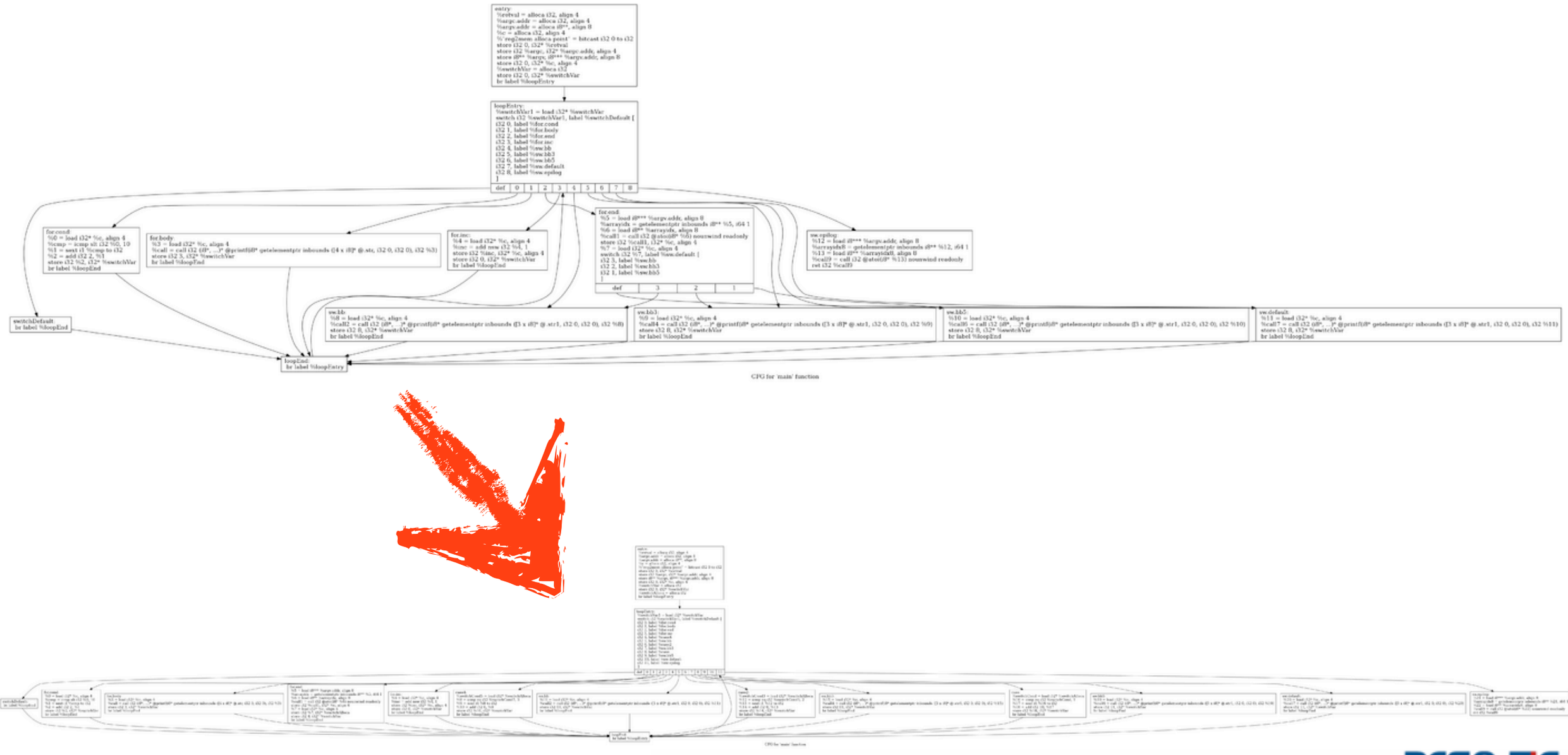
«Code Flattening»

- Cas d'un SWITCH:



«Code Flattening»

- Un cas plus complexe:

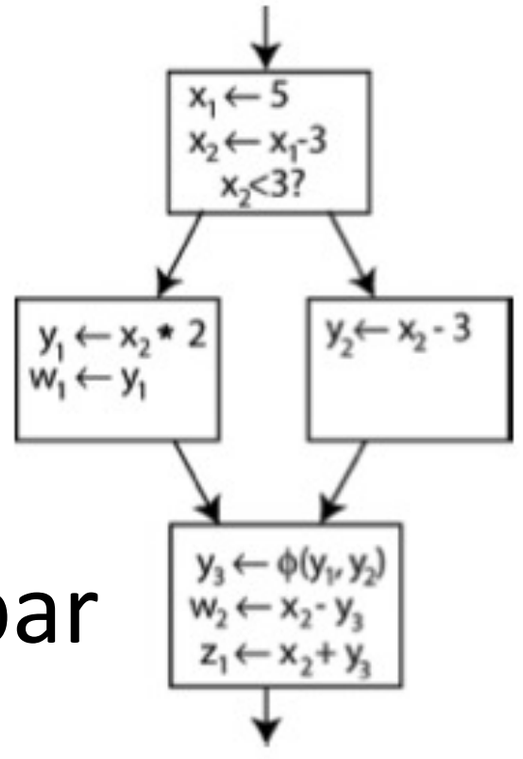


«Code Flattening»

- Un cas vraiment complexe:
  - Une routine comportant 6000+ lignes de code

«Code Flattening»

- Difficultés principales rencontrées
  - Complexité de LLVM et de ses APIs
  - Philosophie SSA («single static assignment») et ses «phi nodes»
    - Remplacement des «phi nodes» par des pointeurs
  - «Debugging» de notre code



Procédures de Test

- Librairie cryptographique `libtomcrypt` ✓
- Librairie graphique ImageMagick ✓

  - 2’729 tests unitaires OK ✓
  - 2 tests échouent sur 28 ✗
- «Test suite» de sqlite
  - 41 tests sur 119’350 échouent ✗

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

En résumé, LLVM-obfuscator fonctionne correctement dans plus de 99.9% des cas ;-)

Performances

- «Flattening de code»
  - Benchmark de `libtomcrypt (-O0)`
    - 30% de perte de rapidité
    - 15% d'augmentation de la taille d'exécutable
  - `-flatten` + `-O3` plus rapide que `-O0`

Performances

- À faire dans le futur
  - Définir une ou plusieurs «polices» de protection (`-B0, -B1, -B2, -B3`) combinant les passes à disposition et les benchmarker sur de nombreux «use-cases»

TODOs

- Résoudre le(s) bug(s) de la passe de «flattening»
- Aplatir les instructions `invoke`
  - Utilisées dans la gestion des blocs `try-catch`
- Ecrire de nouvelles passes (cf. plus loin)

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Protection de binaires

ARM

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

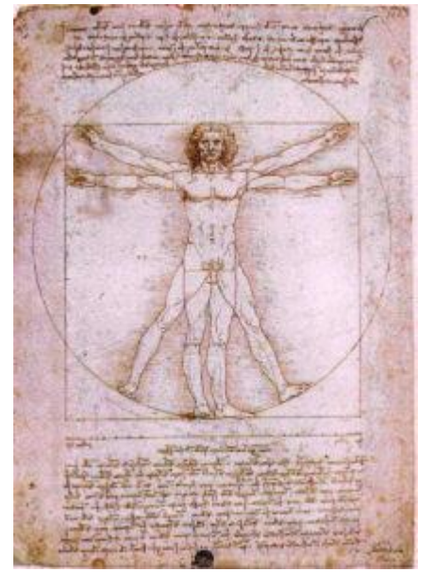
Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Conclusion et Perspectives

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Obfuscation ARM

- Outils similaires utilisés dans le cadre de la sécurité logicielle
- Partagé du savoir avec le groupe «logiciel embarqué»



Obfuscation ARM

- Bonne source de sujets de projets pour les étudiants
  - 2011, *Code obfuscator*, Franzi & Oberson
  - 2012, *Projet Botnet*, Oberson (thèse de master)
  - 2012, Projet de semestre bachelor *Code Obfuscation*
  - 2013, Projet de semestre master *COBRA*, Romanens

LLVM et Obfuscation

- L'outil devient intéressant, et n'est pas très loin d'être «industrialisable».
- Aucun concurrent sérieux «open-source» connu
- Deux seuls compétiteurs «closed-source» connus:

Morpher Crypt the script

ARXAN Protecting the App Economy™

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Obfuscation LLVM
Bilan Pédagogique

- 3 passes d'obfuscation développées
- 1 thèse de bachelor
- 2 projets de semestre master (Bischof & Ruch)
- 2 projets de semestre master à venir sur la virtualisation et le «tamperproofing» (Rinaldini & Wehrli)

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Obfuscation LLVM
Bilan «Dissémination»

- Obfuscator a été présenté lors de 3 workshops locaux, en plus de celui-ci
- AFSWS 2011 & 2012
- 0sec 2011

RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Obfuscation LLVM
Bilan «Académique»

- Contacts pris avec le Prof. Endre Bangerter, BFH
- Projet exploratoire en «désobufscation» en cours, demande d'envergure au SNF dans les domaines de l'obfuscation & désobfuscation

Hes·so University of Applied Sciences Western Switzerland

heig-vd Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Obfuscation LLVM
Bilan «Industriel»

KUDELSKI SECURITY

- Nombreux contacts informels avec Kudelski
- Contacts avec ELCA SA

TECHNOLOGY CONSULTING INNOVATION
ELCA

Obfuscation LLVM

Futur

- Actuellement, l'outil continue d'être développé, grâce à des financements tiers
- Une fois la passe de «flattening» déboguée:
  - Go «open-source»!
- Dissémination scientifique

Hes·so
University of Applied Sciences
Western Switzerland

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Merci pour votre attention !

Q & A