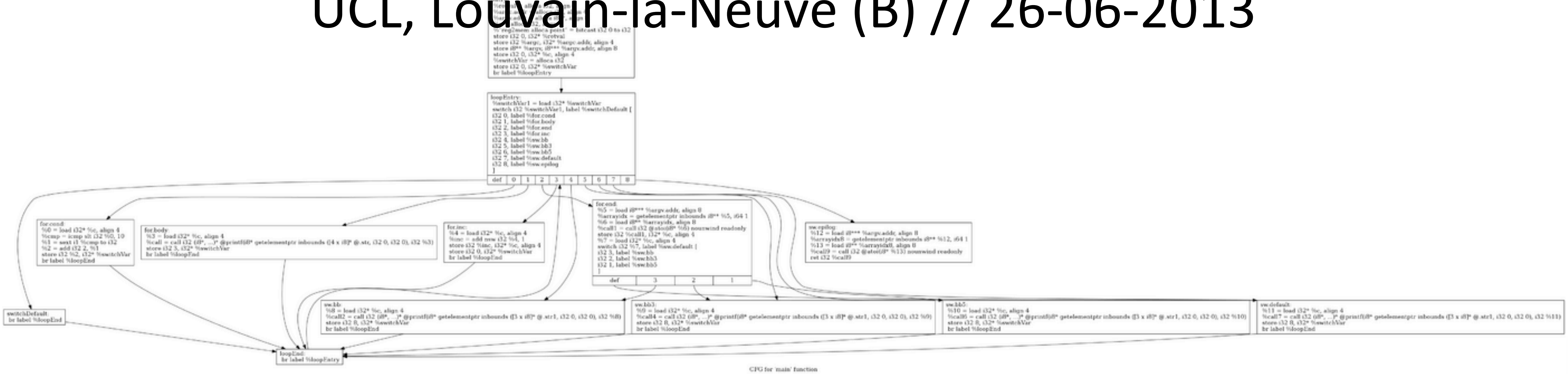
Hes·so
University of Applied Sciences
Western Switzerland

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

LLVM and Code Obfuscation

Pascal Junod

UCL, Louvain-la-Neuve (B) // 26-06-2013



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

Context

Who?

- HEIG-VD, part of the University of Applied Sciences Western Switzerland (HES-SO)
  - Pascal Junod, Julien Rinaldini, Grégory Ruch + several bachelor and master students

What ?

- Everything started with a HES-SO-funded project about software protection.
- Two directions
  - Source-level protection [HEIG-VD]
  - Binary protection [EIA-FR]

How Much ?

- 100 hours (professors)
- 1000 hours (research assistants)
- Until today, the effort has been at least doubled thanks to external funding.
- Forecast until March 2014: 2 man-year

When ?

- First phase: started 12/2010, to 12/2012
- Second phase: started 01/2013, until at least 03/2014

Why?

- Different types of adversaries:

«Black-Box» Adversaries

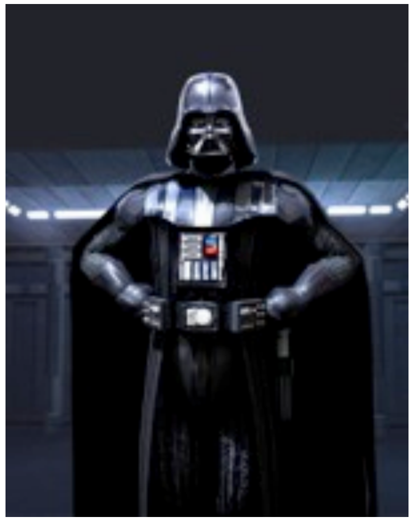
- Play according to the rules (!)
- Interact with components according to the defined APIs
- Adversaries considered in most provably-secure schemes

«Grey-Box» Adversaries

- They are looking to exploit additional «side-channel» information
  - Timing
  - Various types of leakage
  - Faults

«White-Box» Adversaries

- Most powerful types of adversaries
- (Almost) completely master SW/HW
- Can read every memory
- Can disturb every computation at will



«White-Box» Adversaries

- Examples in real-life:
  - DRM circumventing;
  - License management system cracking;
  - Rogue SW «reverse-engineering», IP stealing.

«White-Box» Adversaries

- Just think about this:

```
if (RSA_verify (signature) ==
RSA_VALID_SIGNATURE) {
  // Perform some critical operation
  } else {
   return NOT_AUTHENTICATED
  }
```

«White-Box» Adversaries

- At the assembly level:

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

The whole RSA signature security relies on the fact that this instruction is properly executed or not.

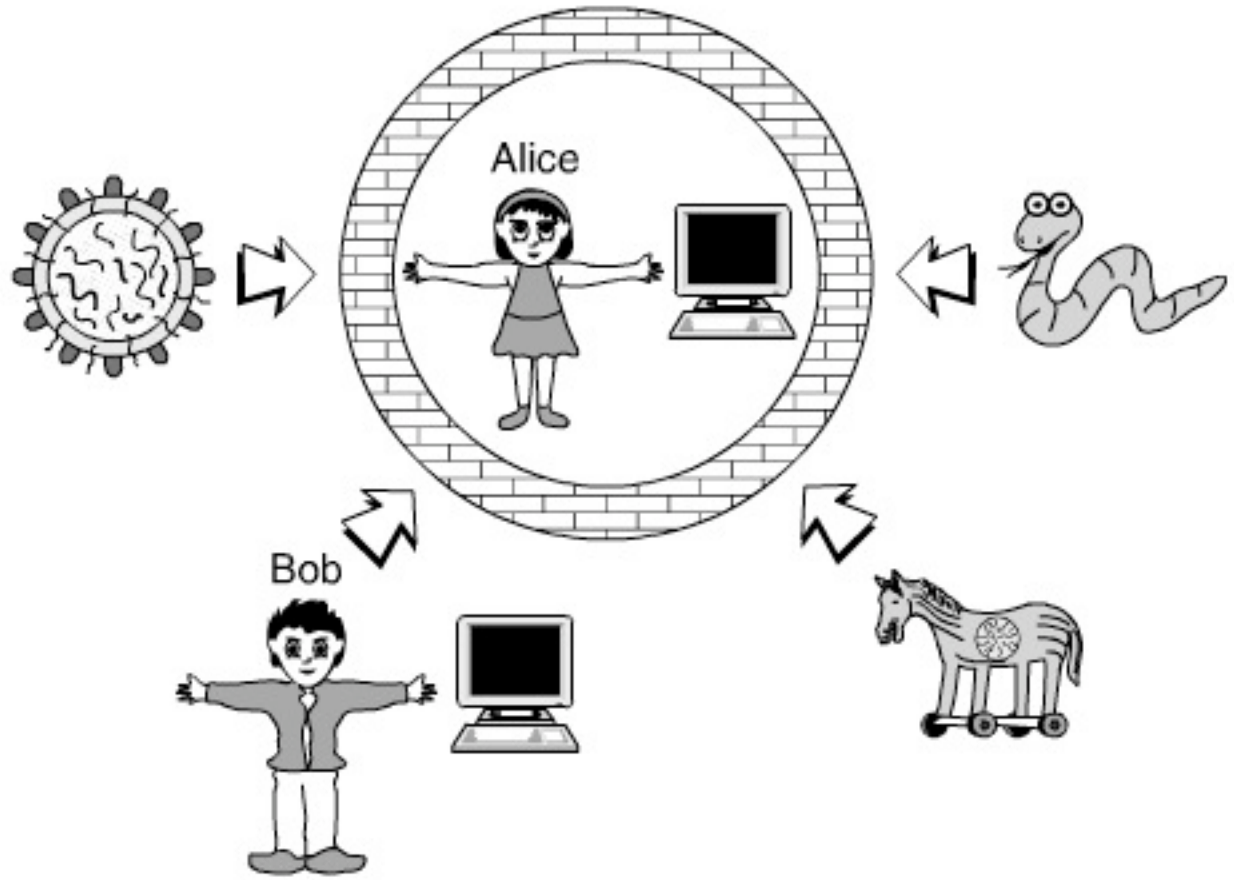
Classical Counter-Measures

- Use of HW as a root of trust:
  - TPM
  - smartcard
  - USB dongle
  - ...

Classical Counter-Measures

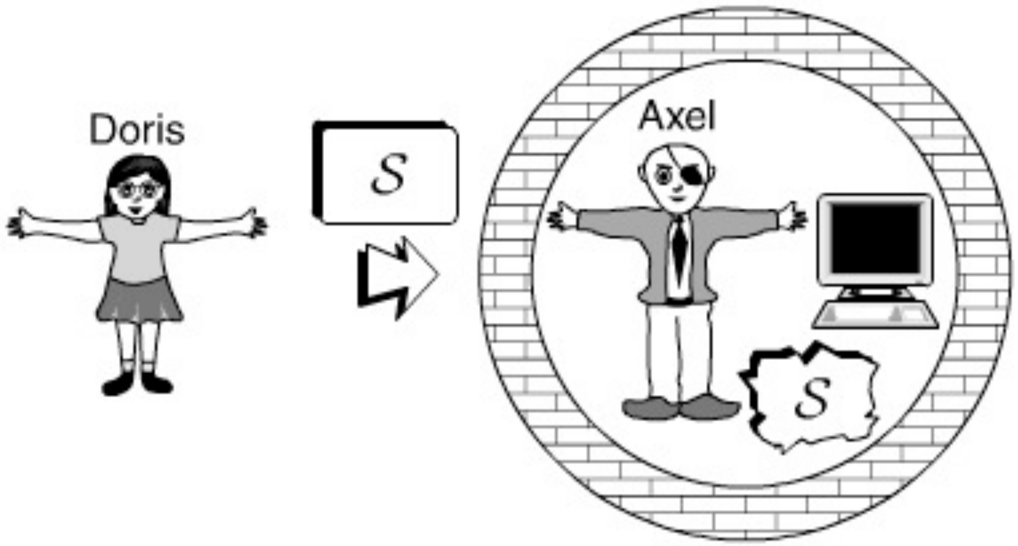
- Problem: a hardware root of trust is not always ...
  - there;
  - sufficiently cheap;
  - flexible;
  - etc.

Software Protection



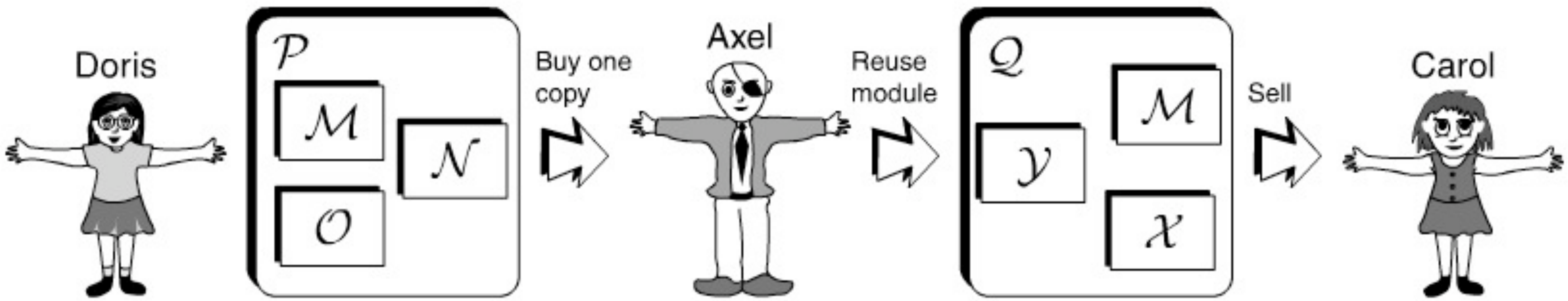
Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009

Software Protection



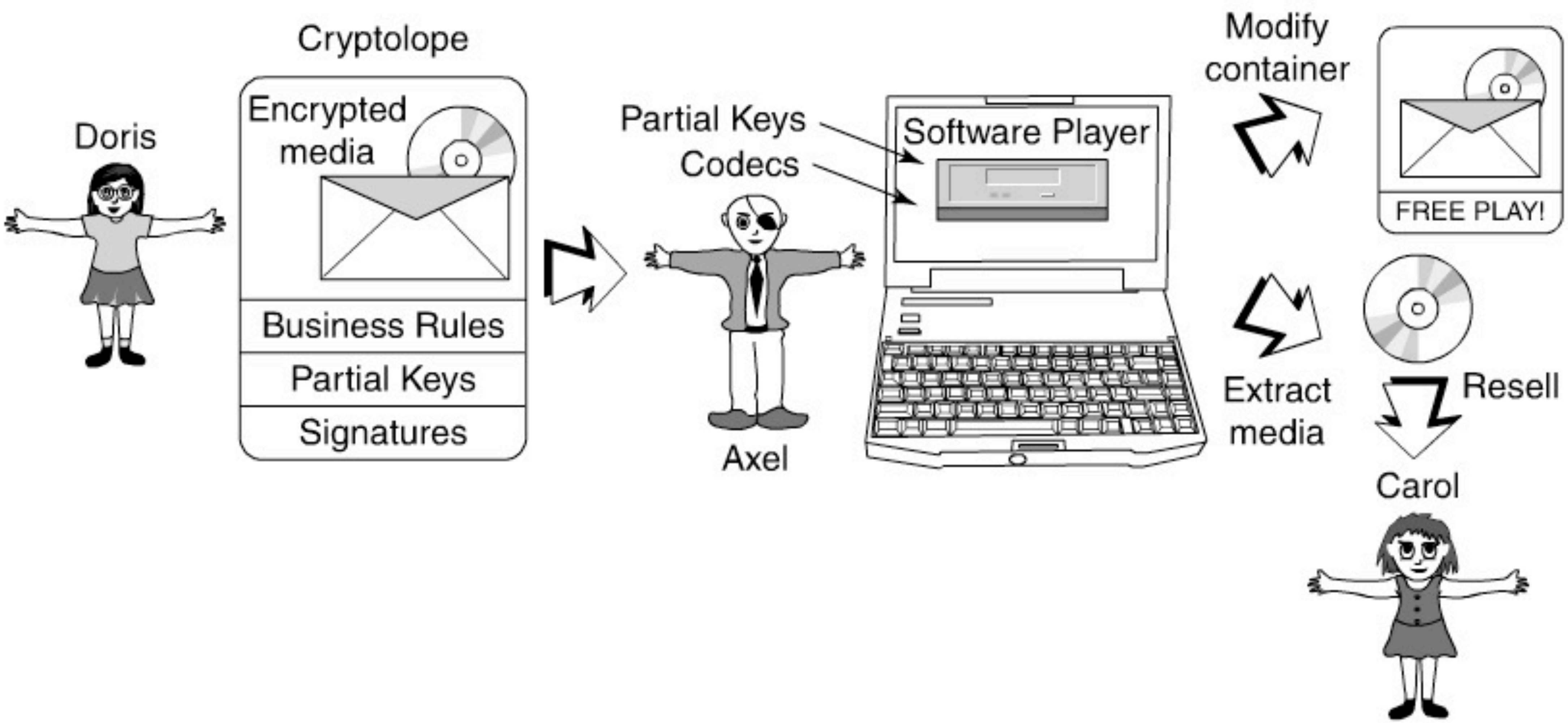
Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009

Rogue «Reverse Engineering»



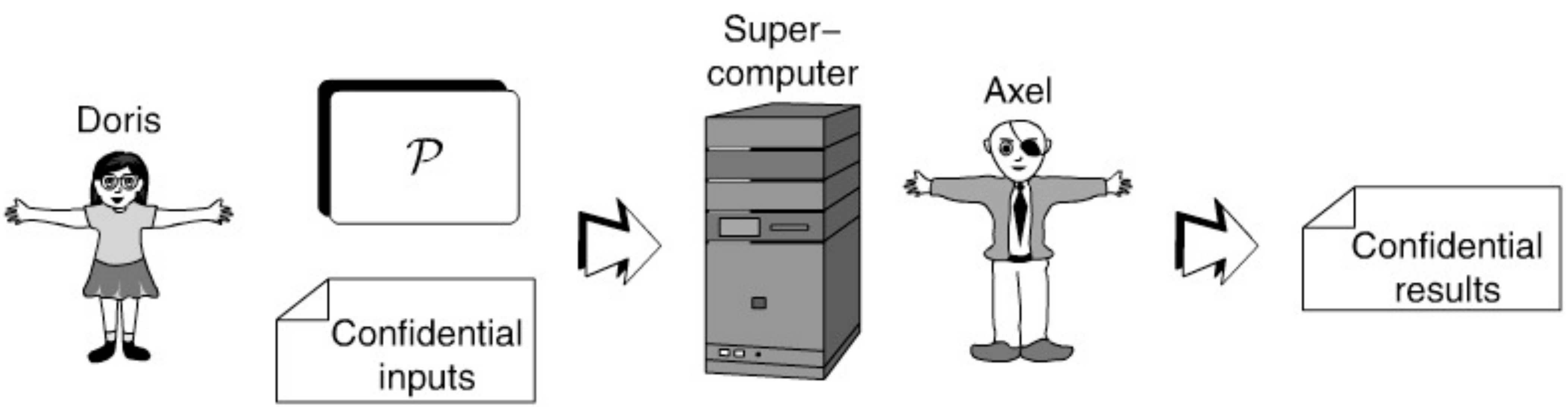
Pictures credit: Collberg, Nagra, «Surreptitious Software», Addison-Wesley, 2009

DRM



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

Cloud Computing



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

Attack scenarios

- «Reverse Engineering», IP extraction, cryptographic secrets extractions.
- Code modification
- Code distribution

Software Protection

Obfuscation

- Render code more difficult to understand

Tamperproofing

- Render code more difficult to modify
- Render code unique

Watermarking

Obfuscation Techniques

Software Protection

- According to Wikipedia, obfuscated software is source or machine code that is hard (or expensive) to understand.

Software Protection

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.

Software Protection

- Even if it is theoretically impossible, let's still try to do it in practice :-)
  - Weaker security assumptions ?
    - The goal consists in rendering the adversary's task more costly, even if does not result in an unpractical effort...

Software Protection

```
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

Software Protection

```
@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
```

This Perl snippet displays «Just another Perl/Unix hacker», several characters at a time, with delays.

Source: Wikipedia

Software Protection

```
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);}
```

An Eratosthenes' sieve

Source: Wikipedia

Software Protection

Software Protection

- Different scenarios
  - Source vs. binary code
  - Supported languages
    - .NET, C#, Java, Javascript, C/C++, (Fortran, Ada, Haskell, Python ?)
  - Costs
    - Size and speed of the resulting code
  - Resistance to reverse engineering

More Advanced Techniques

Packing

Once used for compressing executables, «packers» are often used for software protection purposes.

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

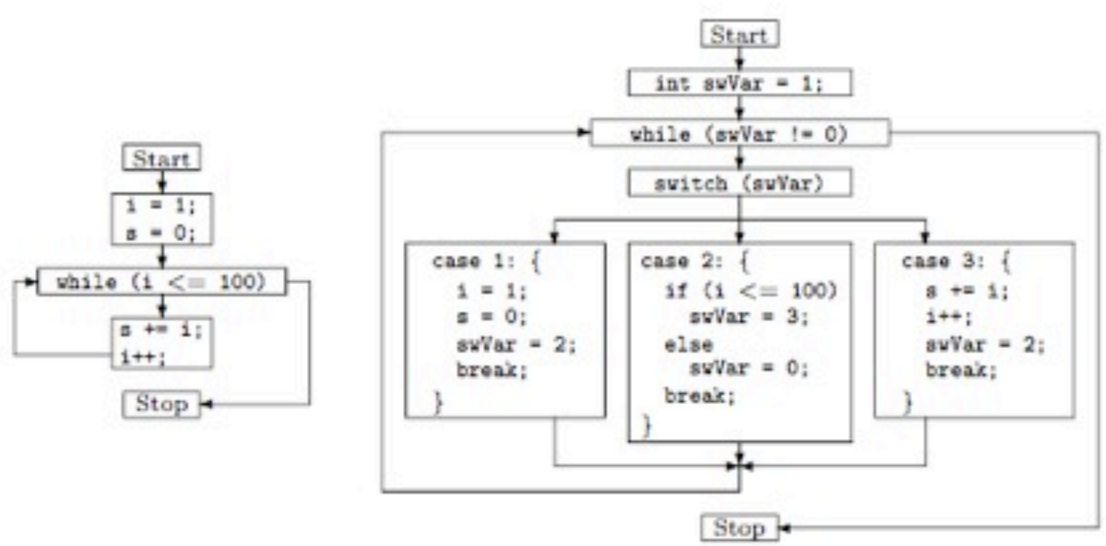
Code Flattening

```
i = 1;
s = 0;
while (i <= 100) {
  s += i;
  i++;
}
```

(a)

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

(b)



Opaques Predicates

- An opaque predicate is a constant boolean expression such that:
  - The developer knows its resulting value;
  - The reverse engineer is supposedly forced to dynamically study its behavior for gaining the same information.
- Force to perform a dynamic analysis («debugging», emulation, ...)

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

- The idea consists in mixing several independent pieces of code.
- Adding junk code;
- Merging procedures;
- Re-splitting in different threads;
- Etc.

Custom Virtualization

- Translate software in a customized and individualized «byte-code», and execute it in a custom VM.
- You can iterate the concept, but be afraid of the resulting performances !
- Use Turing-complete and exotic architectures?
  - `brainfuck` (8 instructions, no operands)
  - `subleq` (1 instruction, 3 operands)
  - ...

Add a Time Dimension

- If a bidirectional and (somewhat) reliable network is available, then one can apply ideas similar to the following:
  - At random times, a server asks the client a checksum of machine code
  - Response delay: less than one second.
  - Allows to detect software tampering.
- Frequently used by the online gaming industry to fight cheating players.

Hes·so
University of Applied Sciences
Western Switzerland

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

Obfuscation with LLVM

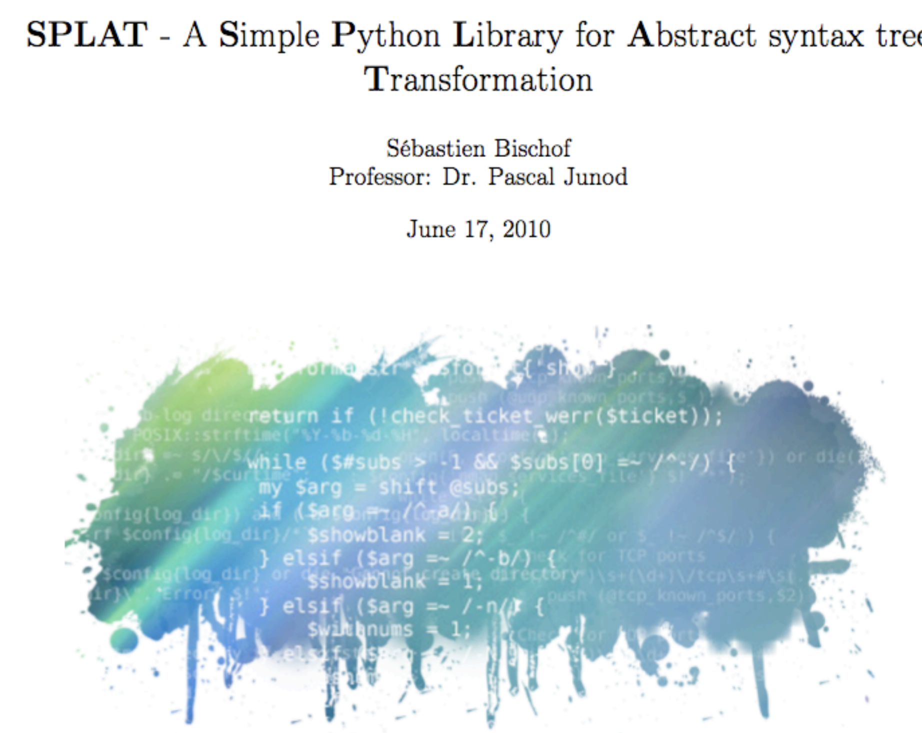
RCSO-TIC
Réseau de Compétences
de Suisse occidentale en technologies
de l'information et de la communication

Motivations

- As a matter of fact, there does not exist open-source SW allowing to obfuscate C or C++ code in an efficient way.
- «Reverse engineering is hard, protection against RE is even harder, so let's face the challenge» !

Abandoned Ideas

- Source parsing, and regeneration of C/C++ code.
- Tool written in Python by Sébastien Bischof, and implementing «code flattening»


SPLAT - A Simple Python Library for Abstract syntax tree Transformation

Sébastien Bischof
Professor: Dr. Pascal Junod

June 17, 2010

Abandoned Ideas

- Major disadvantages of this idea:
  - Very difficult to write a complete and robust parser
  - You have to re-do the job for every language

Abandoned Ideas

- Use an existing «front-end», and add obfuscation capabilities to it
- Work of Grégory Ruch, based on the APIs of Clang, a C/C++ «front-end» of LLVM

Obfuscator - Abstract syntax tree Transformation

Grégory Ruch
Professeur : Dr. Pascal Junod

29 avril 2011

Abandoned Ideas

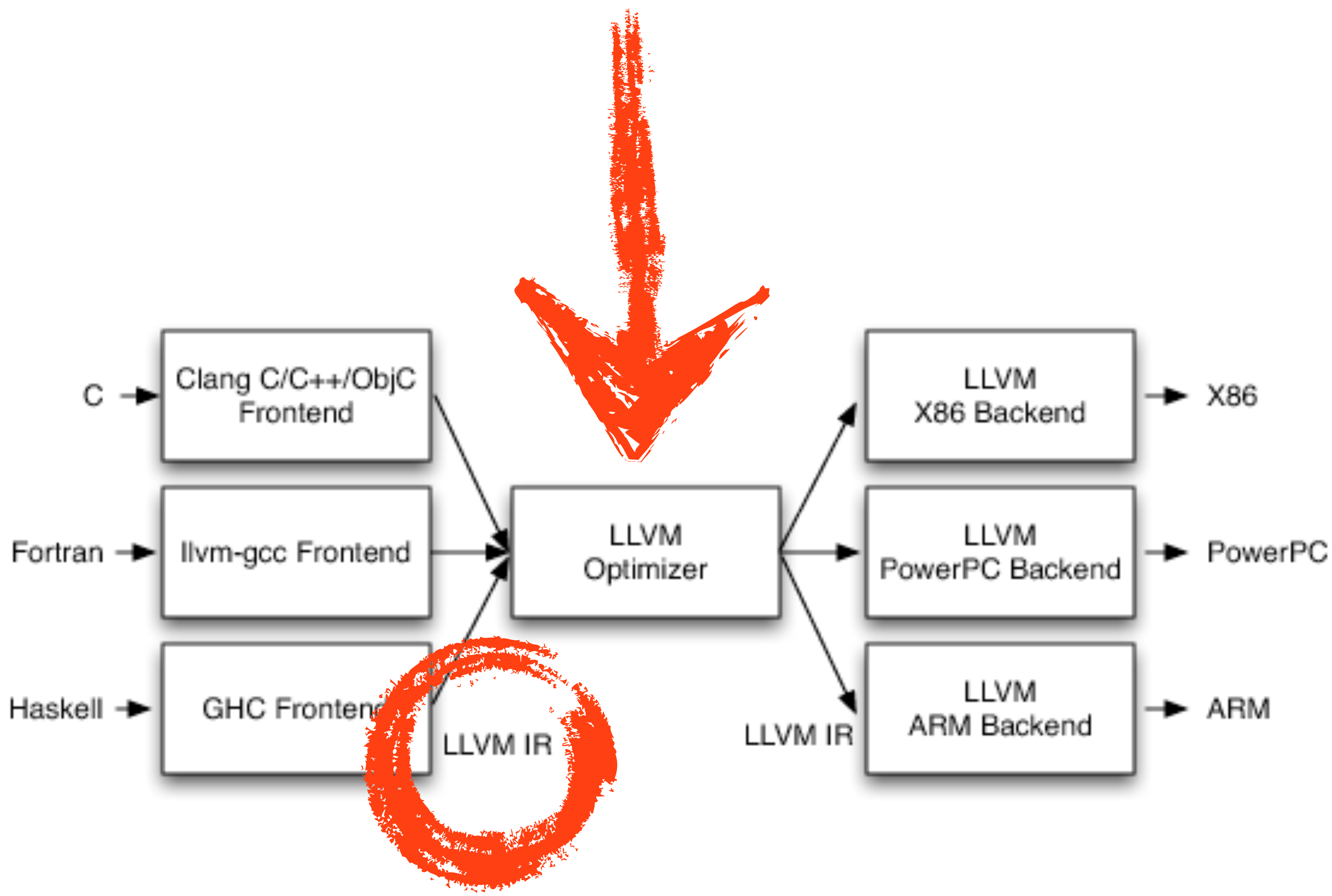
- Major disadvantages of this idea:
  - The APIs of Clang have not really been designed to modify the generated AST.
  - Supporting only C/C++/Objective C.

LLVM

- Complete compilation infrastructure
- Open source project initiated at the University of Illinois in 2000
- Since 2005, the main sponsor is Apple Inc., which hired Chris Lattner
- Very dynamic community
- State-of-the-art software architecture

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
LLVM PowerPC Backend
LLVM ARM Backend
X86
PowerPC
ARM

Listing 3.1 – Simple fonction faisant une addition en C

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

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

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

LLVM and Obfuscation

- LLVM offers a very rich API that is well documented to play with IR code.
- The primary goal was to be able to write language-independent optimization passes in an easy way.
- This approach can be used to perform obfuscation, instead of optimization

LLVM and Obfuscation

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

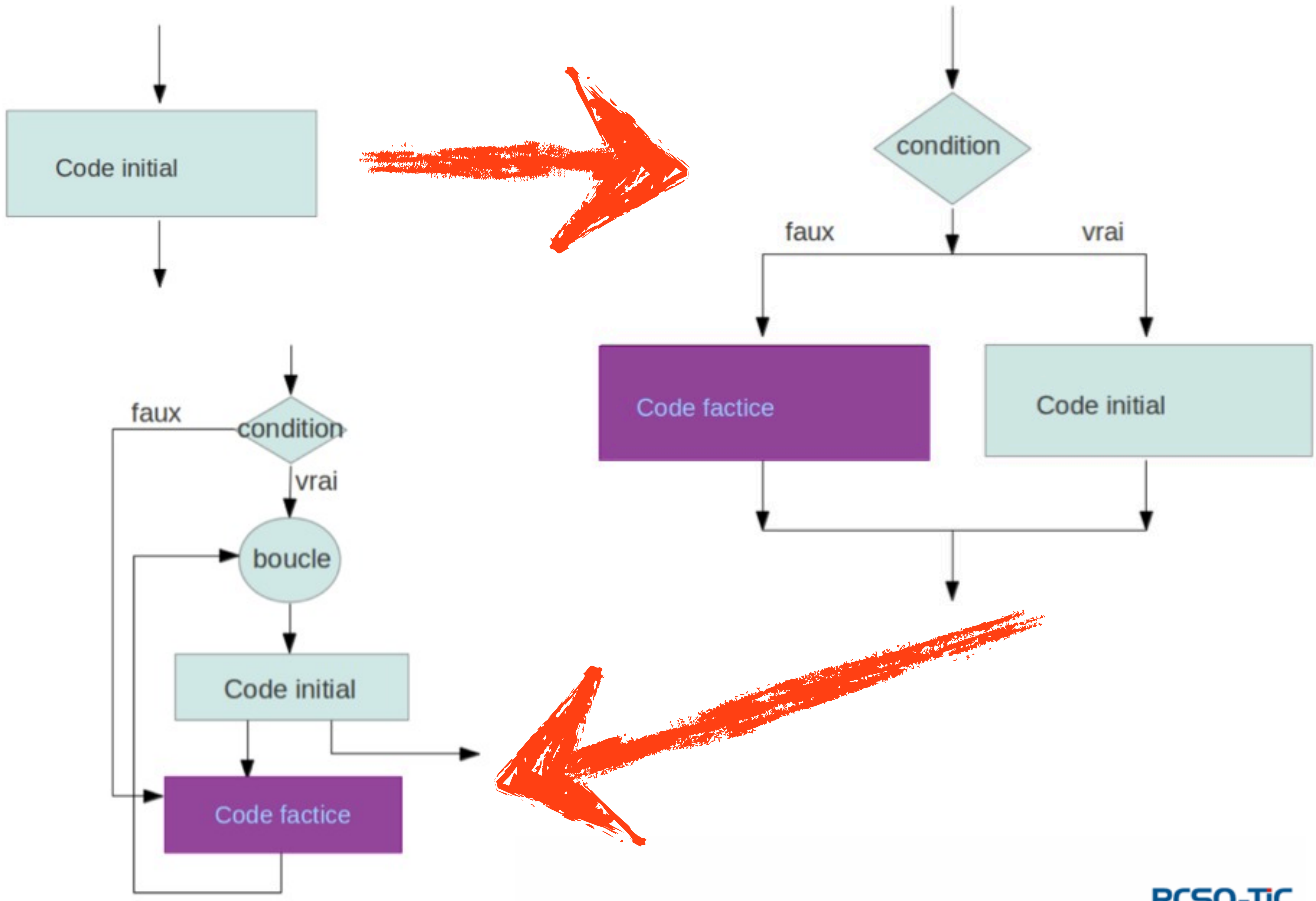
```
$ clang -O3 -B3 -o prog prog.c
```

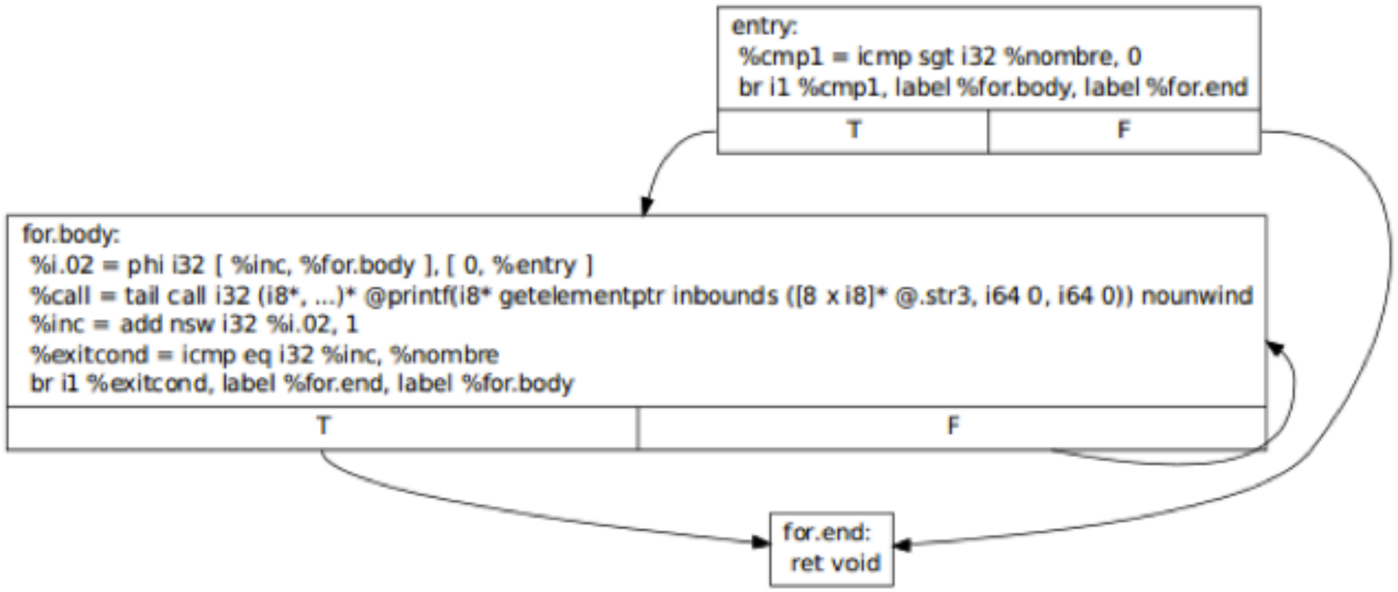
Code Substitution

- First pass written, to familiarize ourselves with the LLVM API
- Replace an instruction by an equivalent expression:
  - `A ^ B = (A & ~B) | (~A & B)`
  - `A + B = A - (-B)`
  - `A+B = (A+R) + (B+R) - 2*R`
  - `...`

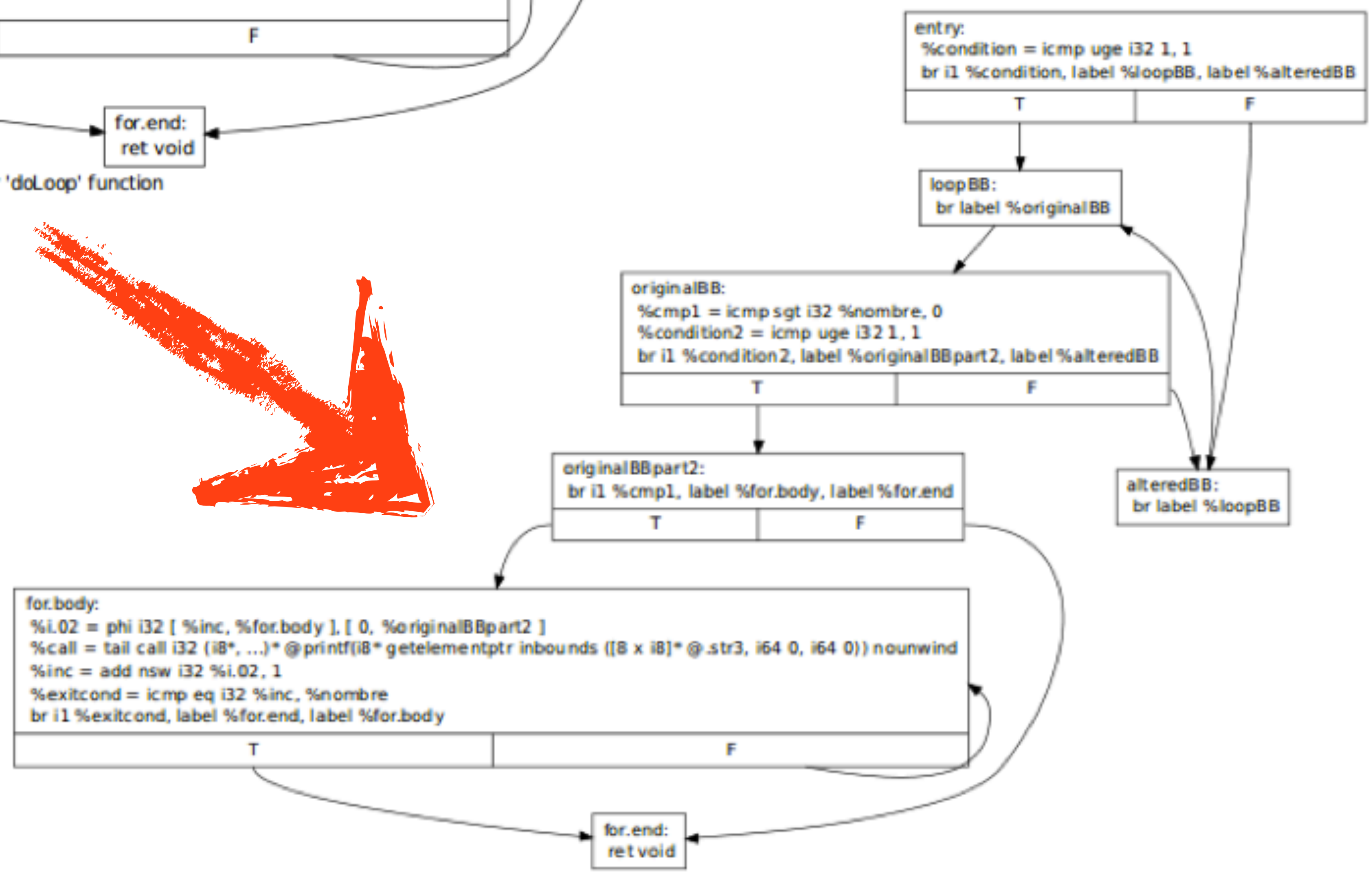
Fake Branches Insertion

- Bachelor thesis of Julie Michielin
- Base ideas:
  - Insert fake branches
  - Render the flow graph irreducible
  - Use opaque predicates


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



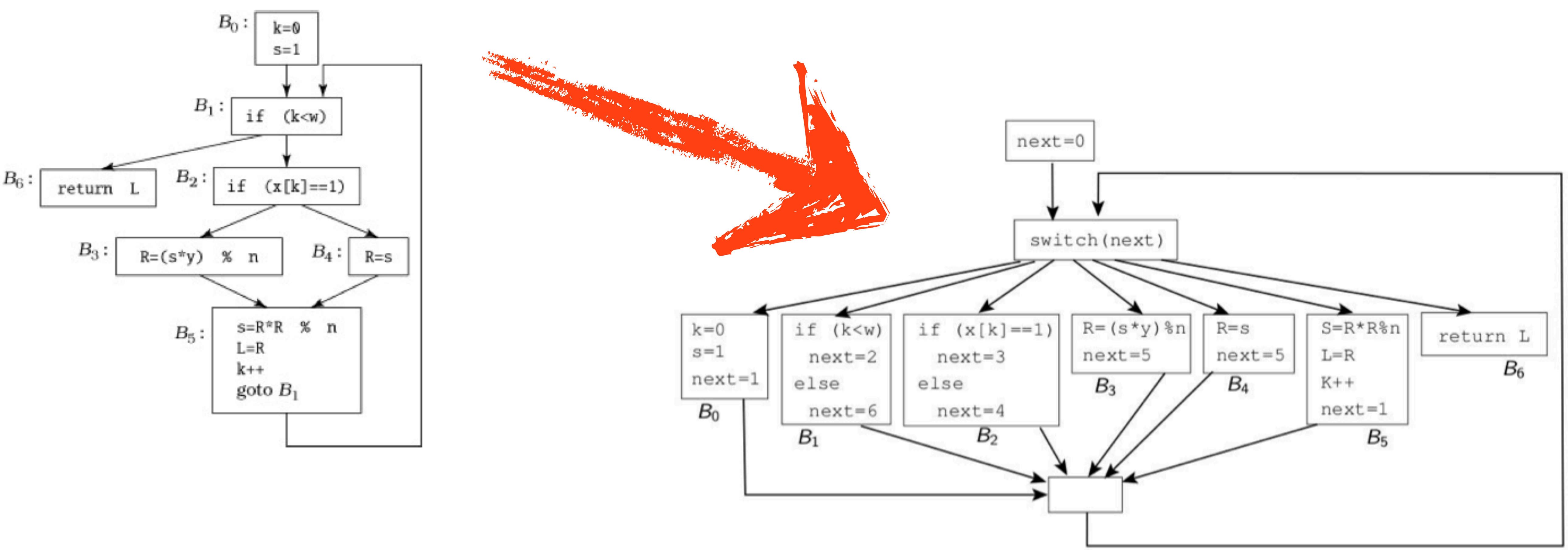
CFG for 'doLoop' function



CFG for 'doLoop' function

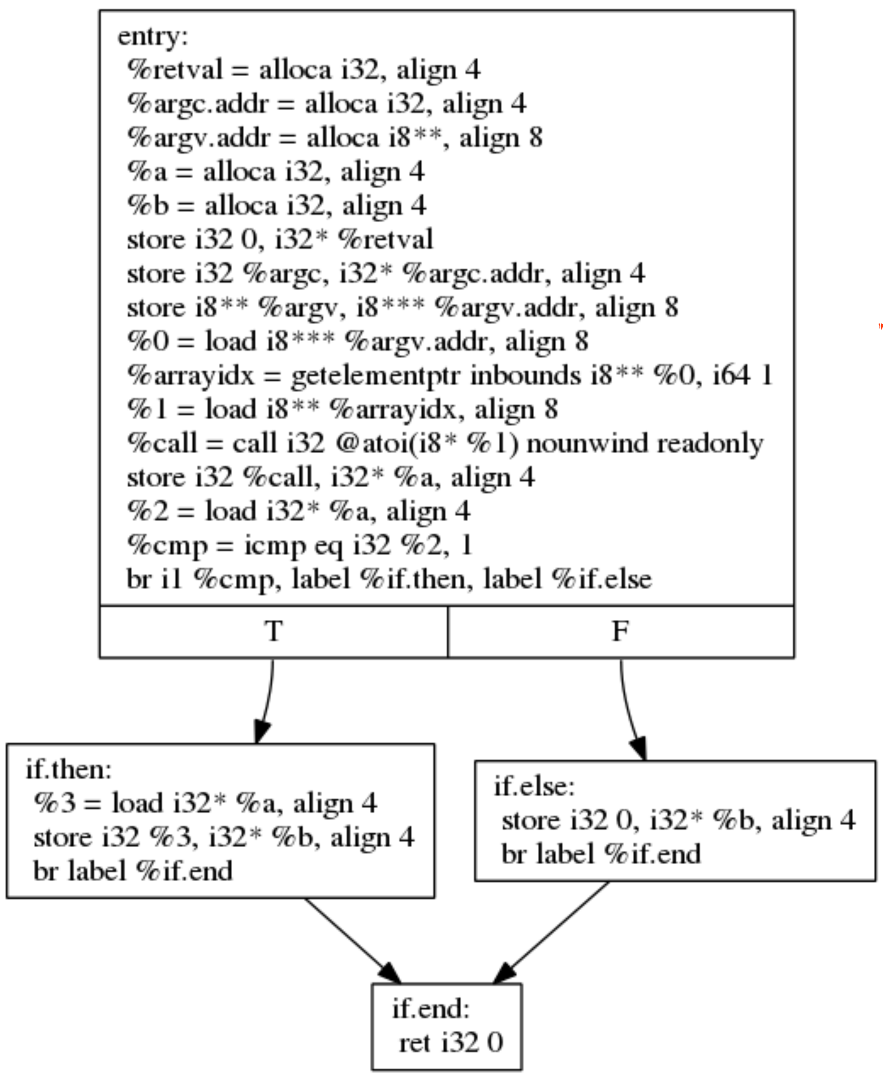
Code Flattening

- By far the most complete working pass as of today

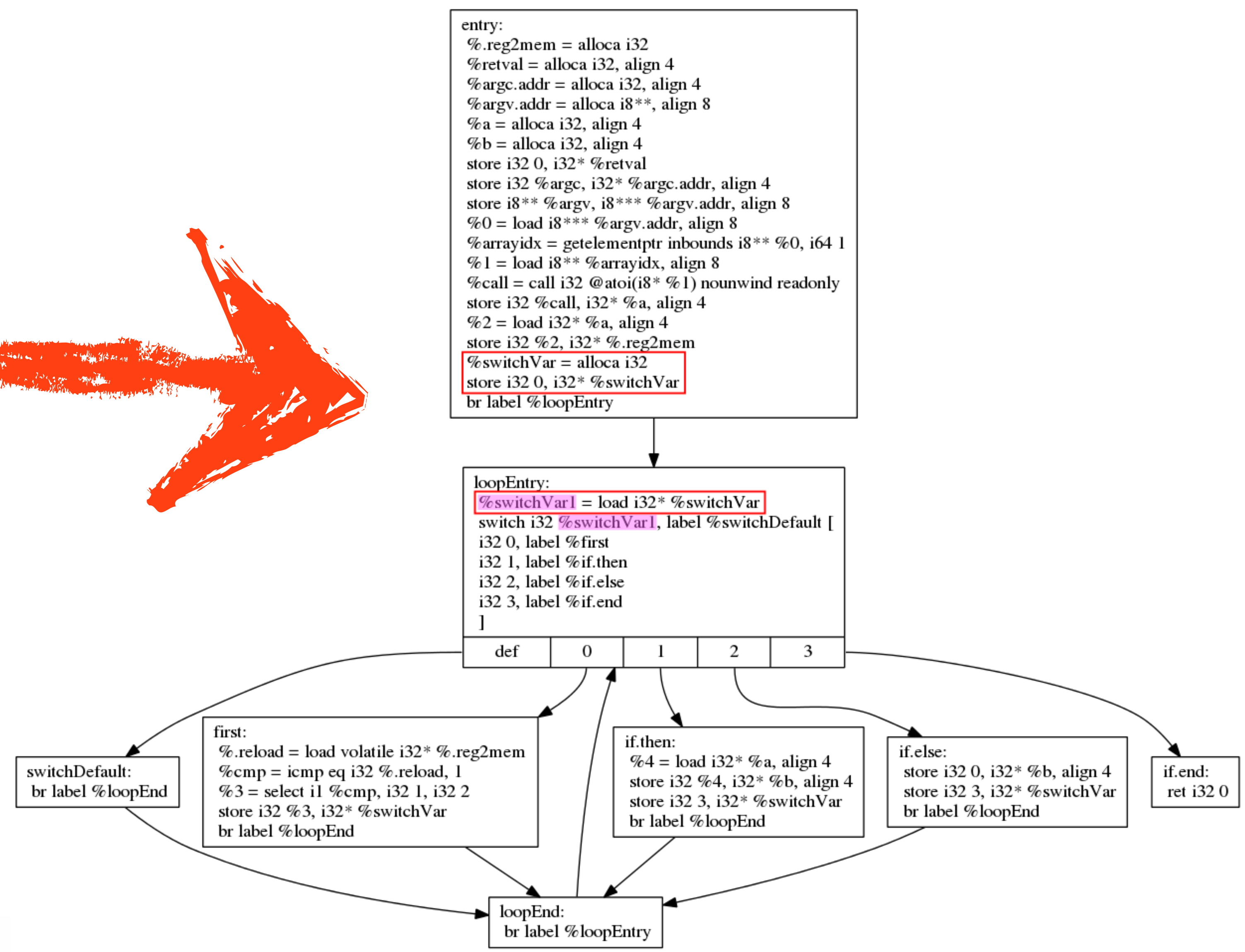


Code Flattening

- Case of IF-THEN-ELSE:



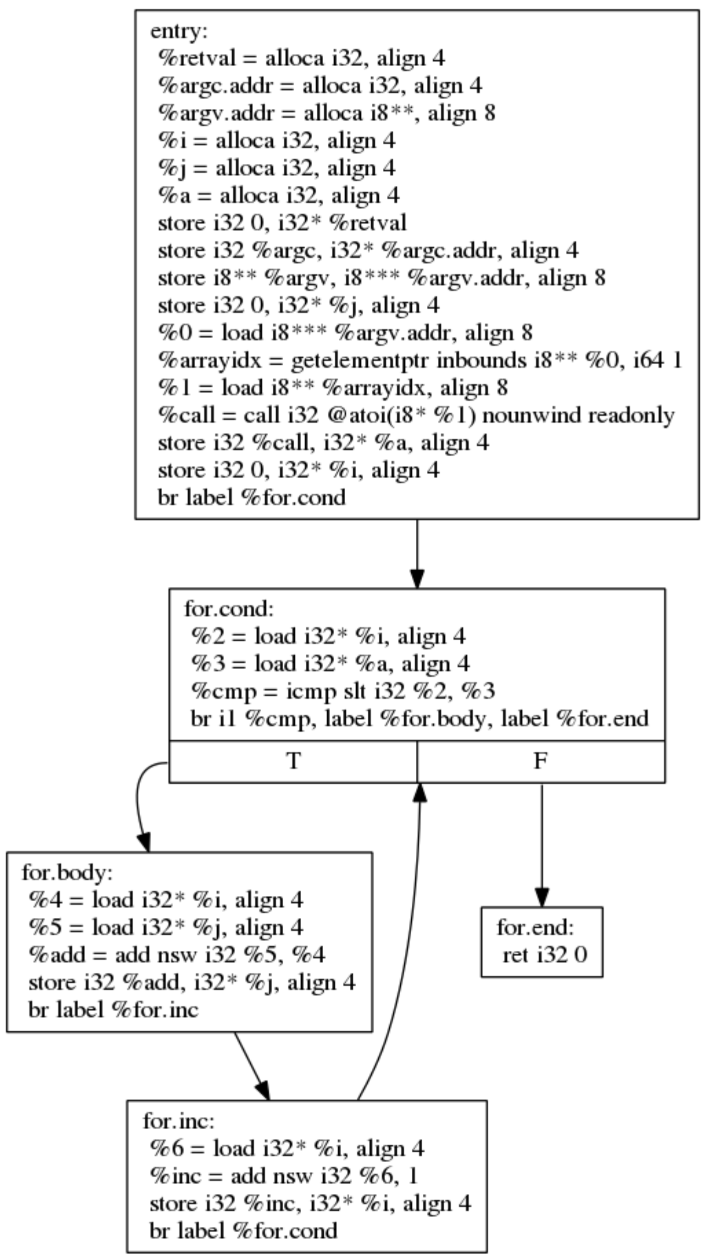
CFG for 'main' function



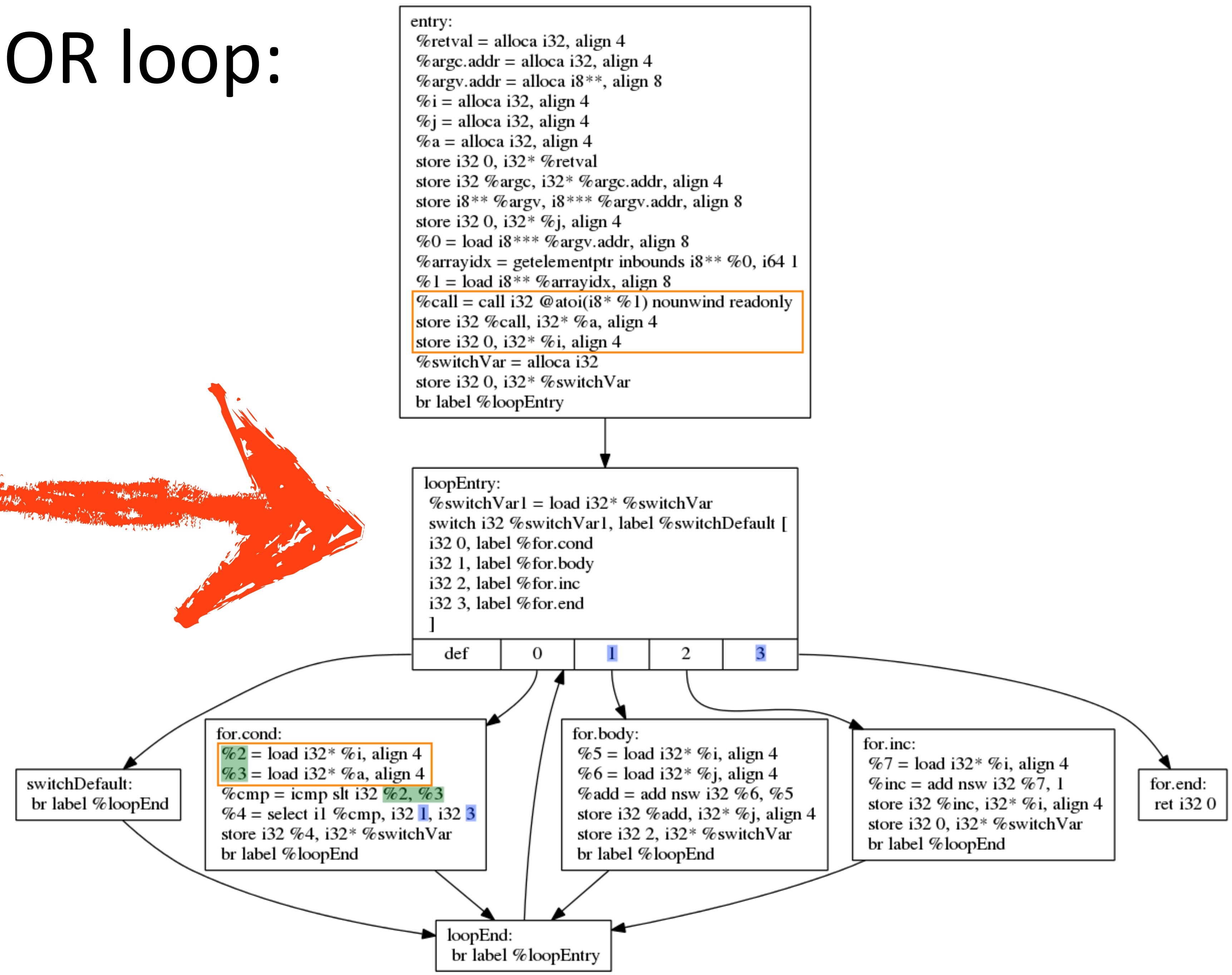
CFG for 'main' function

Code Flattening

- Case of a FOR loop:



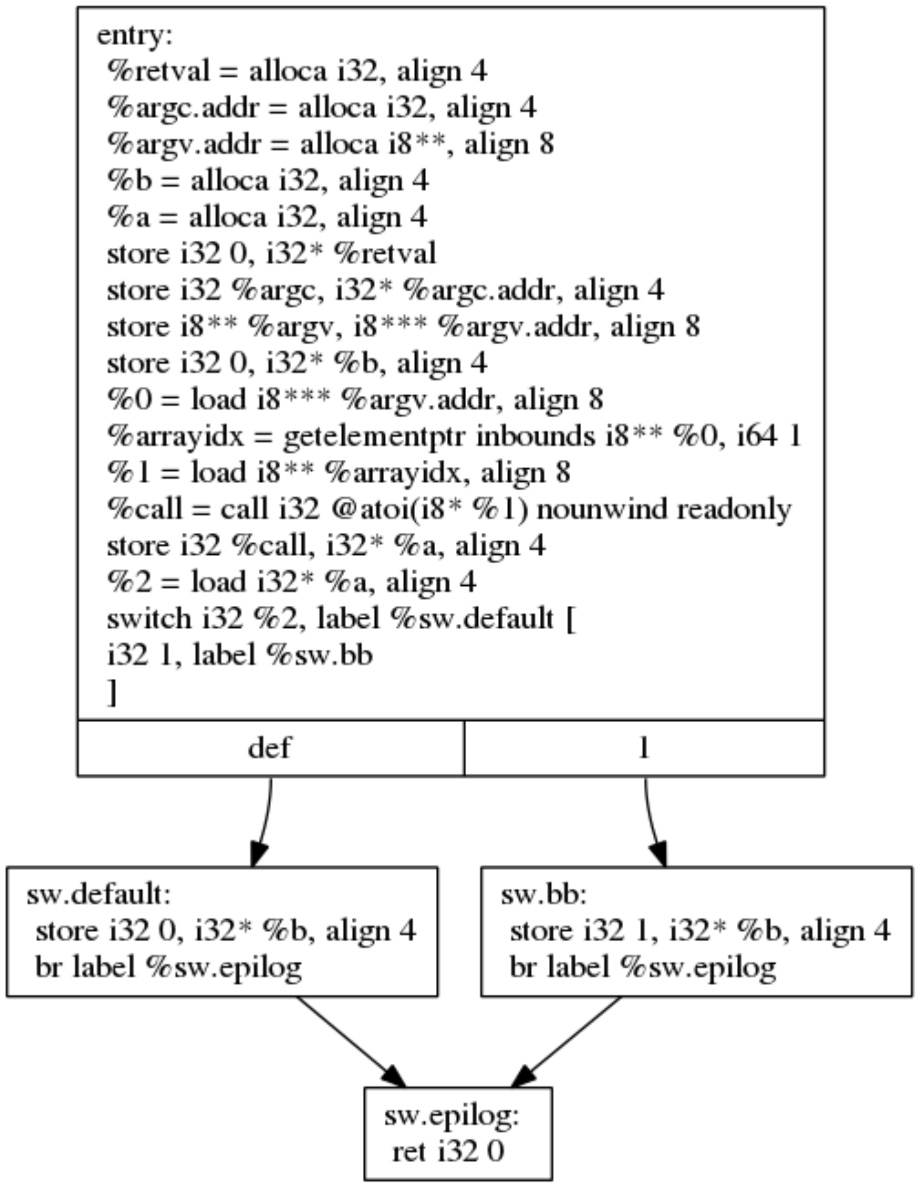
CFG for 'main' function



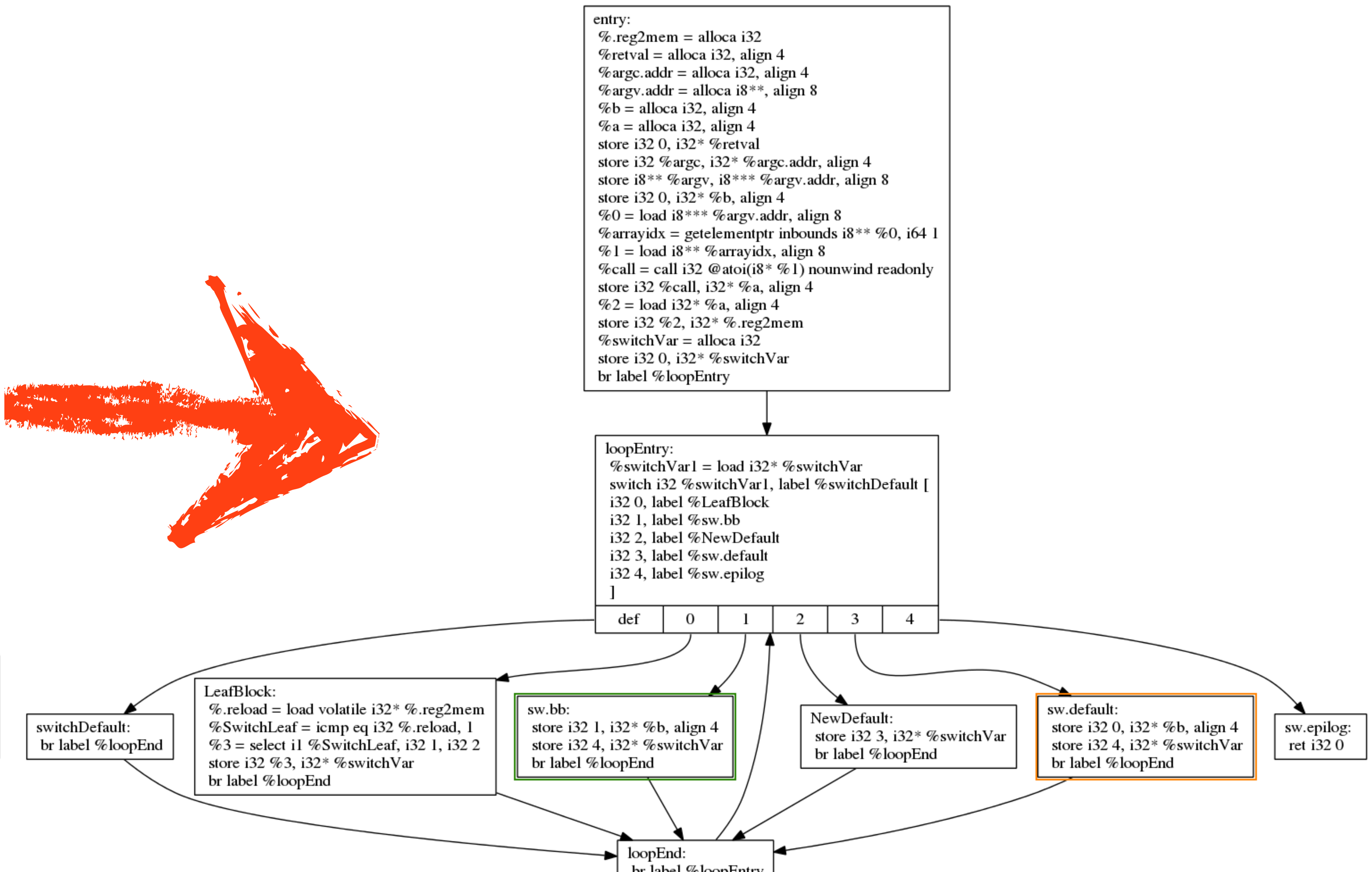
CFG for 'main' function

Code Flattening

- Case of a SWITCH:



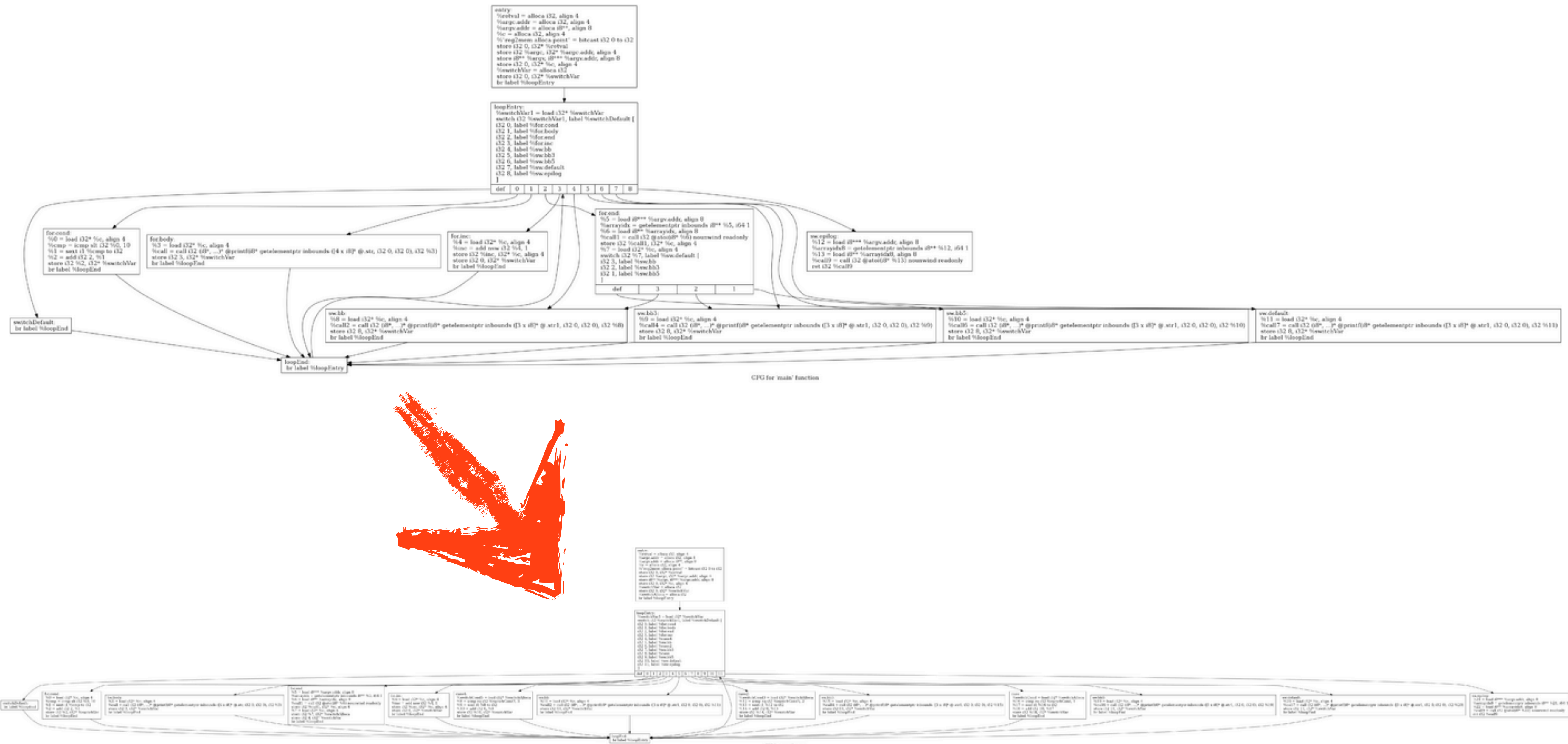
CFG for 'main' function



CFG for 'main' function

Code Flattening

- A more complex case

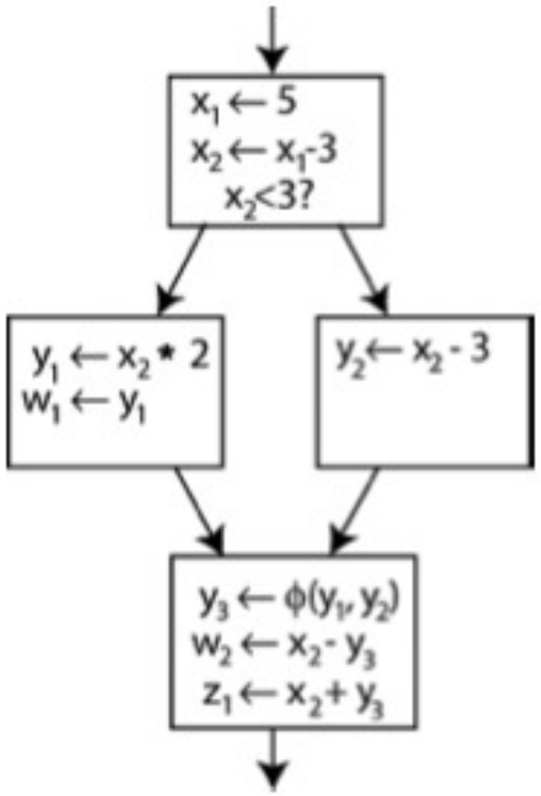


Code Flattening

- A really complex case:
  - One single routine involving found in ImageMagick involving 6000+ lines of C

Code Flattening

- Main encountered difficulties
  - Complexity of LLVM and of its APIs
  - SSA philosophy («single static assignment»), and its «phi nodes»
  - «Debugging» of our code



Test Procedures

- Library `libtomcrypt` ✓
- Library ImageMagick ✓
- Test suite of MySQL
  - 2'729 unitary tests OK ✓
  - 2 tests failing on 28 ~
- Test suite of sqlite
  - 41 tests out of 119'350 failing ~

Performances

- Code flattening
  - Benchmark of `libtomcrypt (-O0)`
    - 30% less speed
    - 15% size increase
  - `-flatten` + `-O3` faster than `-O0`

TODOs

- Identify the remaining bug in our flattening pass
- Flatten the `invoke`
  - Used for handling the `try-catch`

Work in Progress

- Procedures merging pass
- Working at the `x86` back-end level
  - Several technique to thwart faults
  - Code spaghettization
  - Anti-debugging tricks insertion
- Tamper-proofing
- Buffer encryption, code packing
- Custom virtualization using the LL**VM**

Go Open-Source!

github
SOCIAL CODING

Thank you!

Questions welcome!