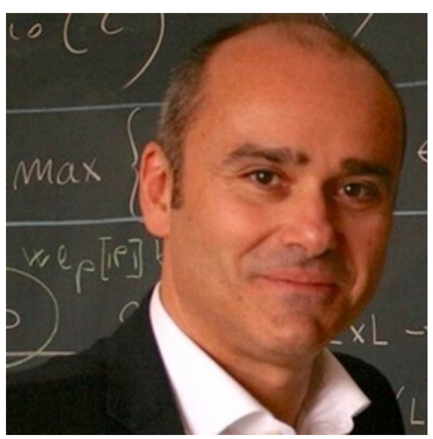
Towards a formally verified obfuscating compiler

Sandrine Blazy

joint work with Roberto Giacobazzi and Alix Trieu



IFIP WG 1.9/2.15, 2015-07-16

Background: verifying a compiler

Compiler + proof that the compiler does not introduce bugs

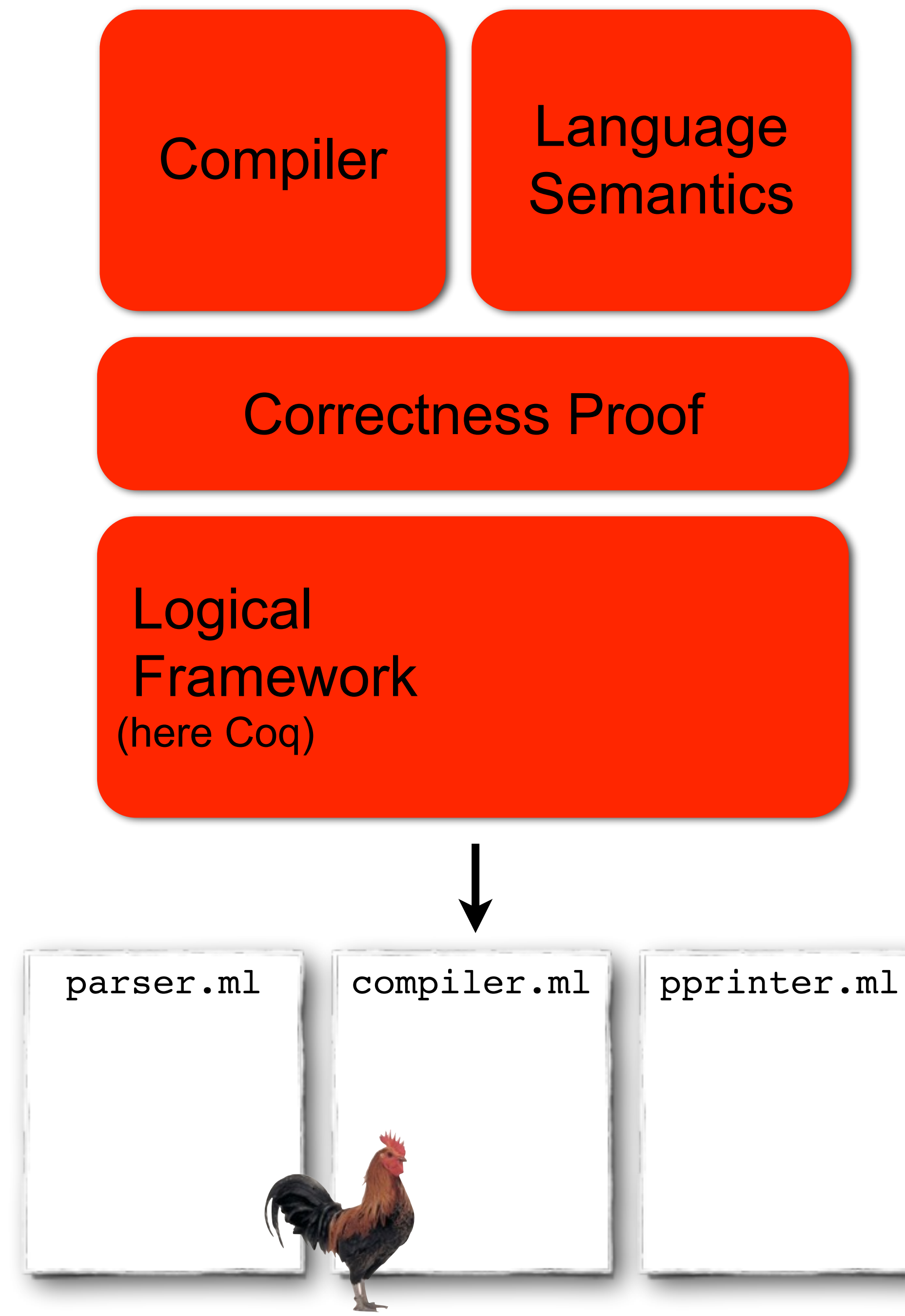
CompCert, a moderately optimizing C compiler usable for critical embedded software

- Fly-by-wire software, Airbus A380 and A400M, FCGU (3600 files):
  mostly control-command code generated from Scade block diagrams + mini. OS

- Formal verification using the Coq proof assistant

Methodology

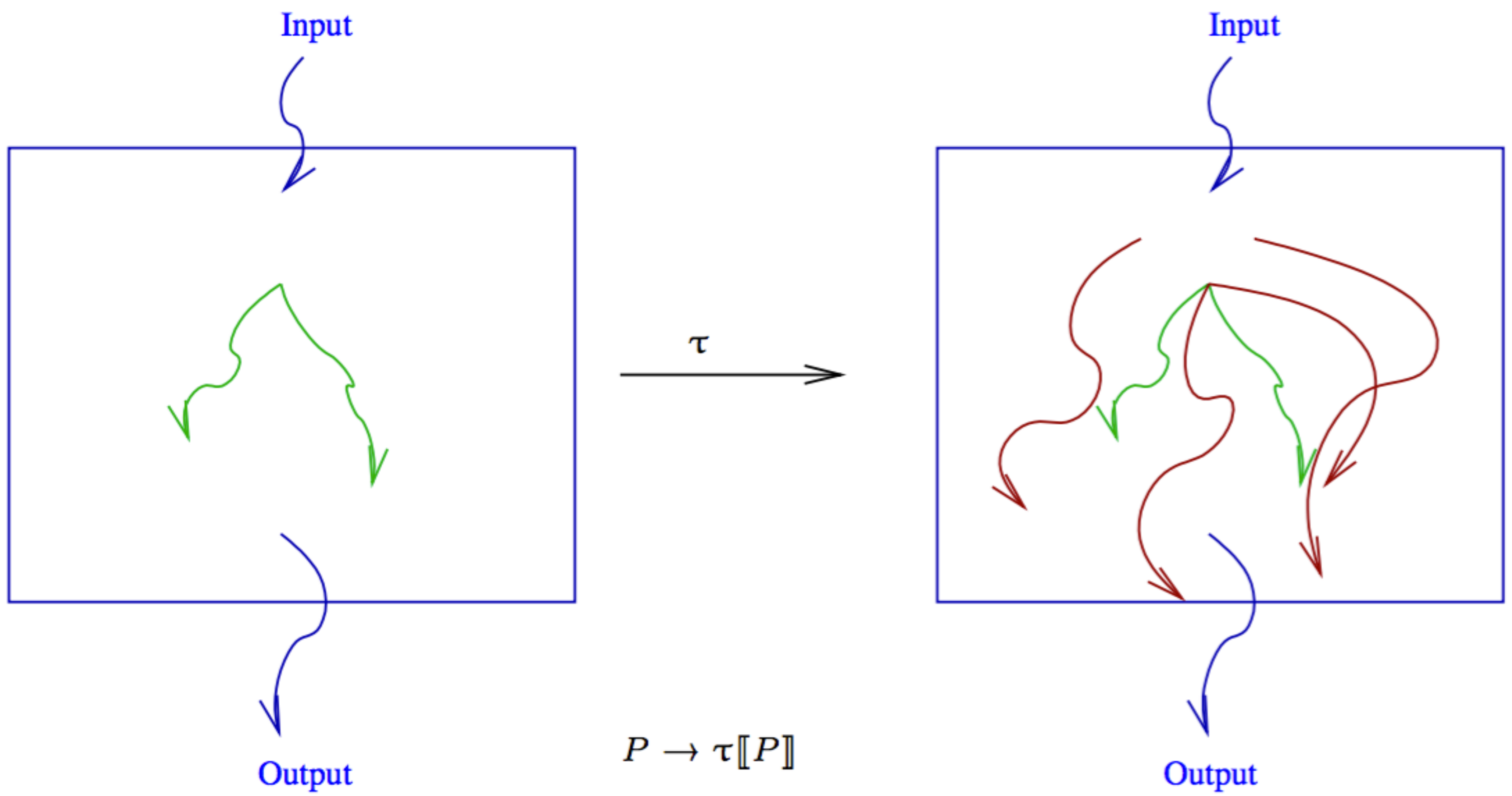
- The compiler is written inside the purely functional Coq programming language.
- We state its correctness w.r.t. a formal specification of the language semantics.
- We interactively and mechanically prove this.
- We decompose the proof in proofs for each compiler pass.
- We extract a Caml implementation of the compiler.



Let’s add some program obfuscations
at the C source level

and prove that they preserve
the semantics of C programs.

Program obfuscation



Recreational obfuscation

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_
            _-_-_-_
}
```

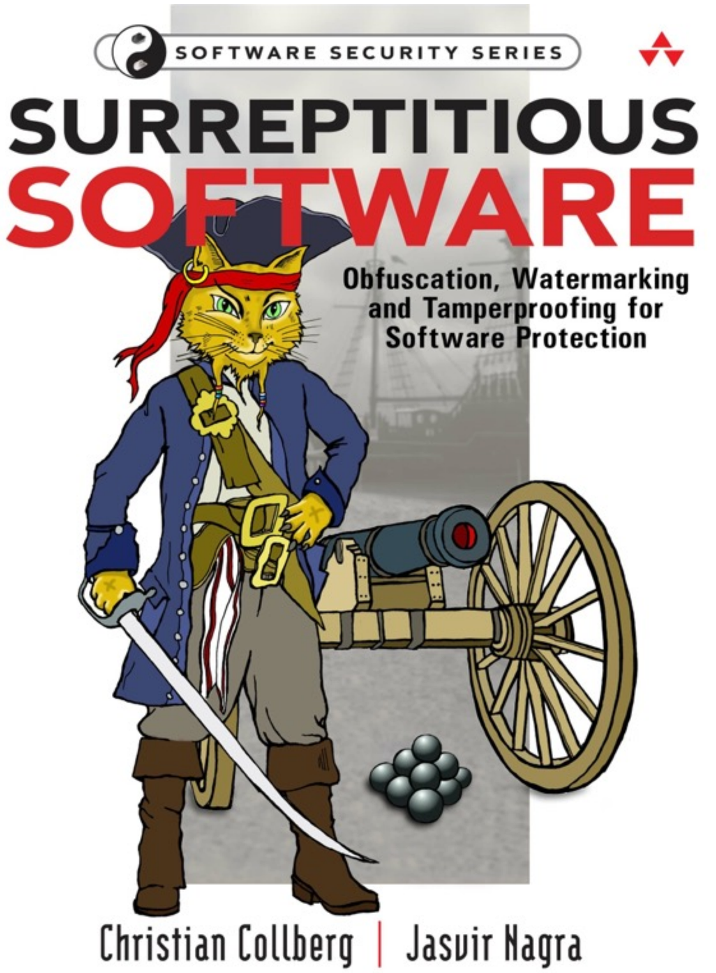
Winner of the 1988 International Obfuscated C Code Contest

Program obfuscation

Goal: protect software, so that it is harder to reverse engineer
→ Create secrets an attacker must know or discover in order to succeed

- Diversity of programs
- A recommended best practice

Program obfuscation: state of the art

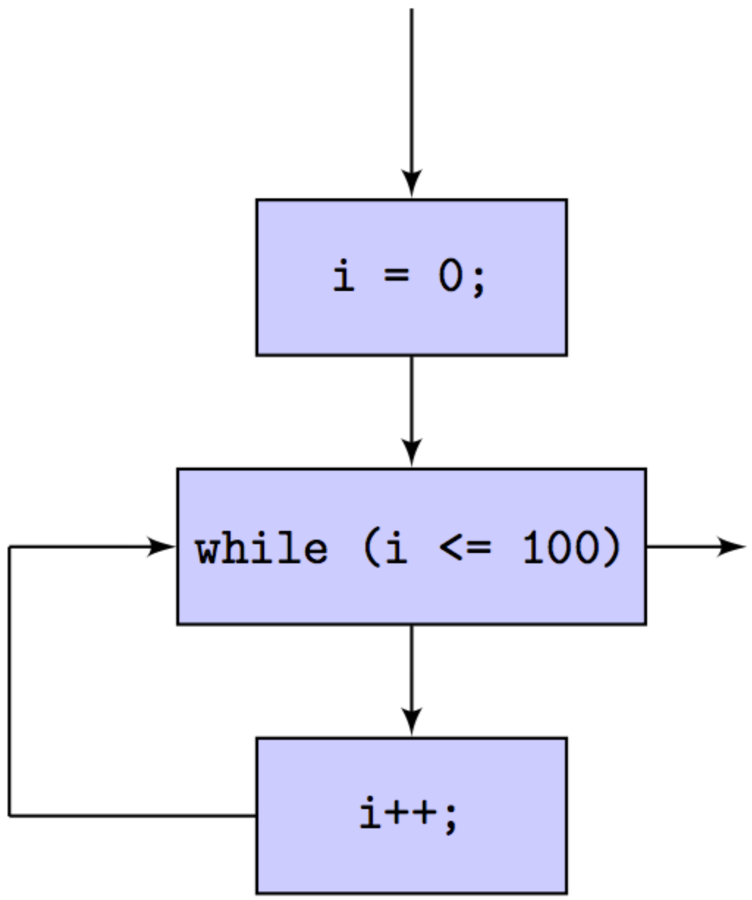


- Trivial transformations: removing comments, renaming variables
- Hiding data: constant encoding, string encryption, variable encoding, variable splitting, array splitting, array merging, array folding, array flattening
- Hiding control-flow: opaque predicates, function inlining and outlining, function interleaving, loop transformations, control-flow flattening

```
int original (int n) {
    return 0; }
```

```
int obfuscated (int n) {
    if ((n+1)*n%2==0)
        return 0;
    else return 1;}
```

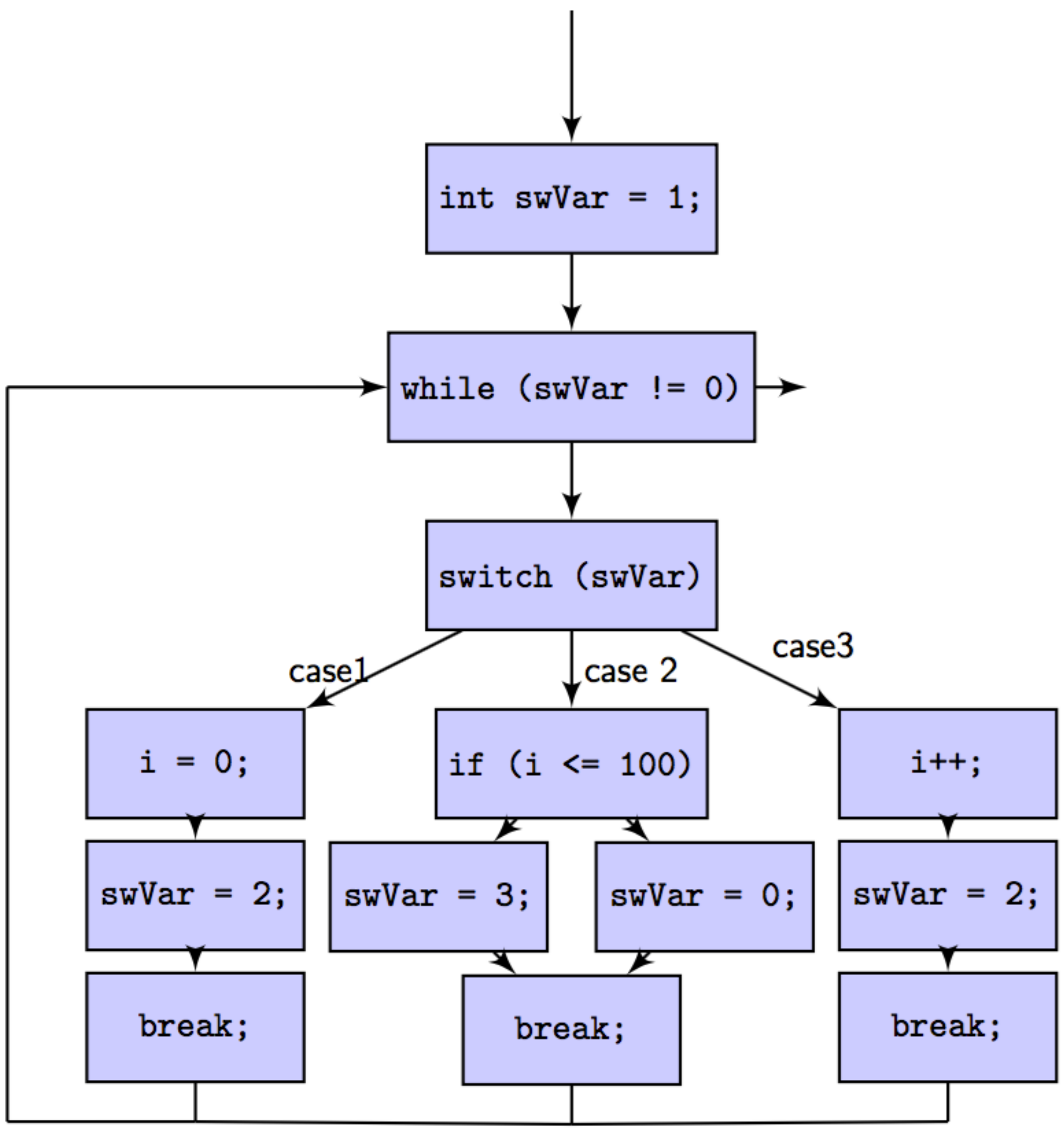
Program obfuscation: control-flow graph flattening




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

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

Program obfuscation: control-flow graph flattening




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

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

Obfuscation: issues

- Fairly widespread use, but cookbook-like use

No guarantee that program obfuscation is a semantics-preserving code transformation.

→ Formally verify some program obfuscations

- How to evaluate and compare different program obfuscations ?

Standard measures: cost, potency, resilience and stealth.

→ Use the proof to evaluate and compare program obfuscations
The proof reveals the steps that are required to reverse the obfuscation.

Formal verification of program obfuscation

Formalizing program obfuscations

- A simple imperative language
  (with arithmetic expressions, boolean expressions and statements)

  Judgements of the big-step semantics
  $\vdash M, a : v$ $\quad\quad$ $\vdash M, b : v$ $\quad\quad$ $\vdash M, s \to M'$

    - Proofs of semantic preservation, mechanized in Coq,
      involving different proof patterns

    - Formalization with Why3

- The Clight language of the CompCert compiler

  Proofs of semantic preservation, mechanized in Coq

Which obfuscations ?

1. Opaque predicates (e.g. $a^2-1 \neq b^2$)
   - Given b_p, every boolean expression becomes `b & bp`.
2. Integer encoding
   - Given O_{val}, every integer constant `n` becomes O_{val}`(n)`, eg. `n+6`.

More generally, we specify 3 functions: O_{aexp}, O_{bexp}, and O_{stmt} and the corresponding deobfuscations functions D_{aexp}, D_{bexp}, and D_{stmt}.

Remark: they can be only axiomatized.

3. Control-flow flattening

A first obfuscation: opaque predicates

We state and prove the semantic preservation of the obfuscation.

- The proof proceeds by induction on the corresponding execution relation (or by structural induction on a syntactic term).

Theorem obf-bexp-correct:

$\forall M, b, v, \quad \vdash M, b : v \quad \Leftrightarrow \quad \vdash M, O_{bexp}(b) : v$

Theorem obf-stmt-correct:

$\forall M, s, M', \quad \vdash M, s \rightarrow M' \quad \Leftrightarrow \quad \vdash M, O_{stmt}(s) : M'$

A second obfuscation: integer encoding

Arithmetic expression obfuscation:

$$\begin{aligned}
\mathcal{O}_{aexp}(n) &= \mathcal{O}_{val}(n)\\
\mathcal{O}_{aexp}(id) &= id\\
\mathcal{O}_{aexp}(a_1 \odot a_2) &= \mathcal{O}_{aexp}(a_1) \odot \mathcal{O}_{aexp}(a_2)\\
&\quad \odot \in \{+,-,*,/\}
\end{aligned}$$

Boolean expression obfuscation:

$$\begin{aligned}
\mathcal{O}_{bexp}(\texttt{TRUE}) &= \texttt{TRUE}\\
\mathcal{O}_{bexp}(\texttt{FALSE}) &= \texttt{FALSE}\\
\mathcal{O}_{bexp}(a_1 \circ a_2) &= \mathcal{O}_{aexp}(a_1)\, o\, \mathcal{O}_{aexp}(a_2)\\
&\quad \circ \in \{==,<=\}\\
\mathcal{O}_{bexp}(b_1 \,\&\&\, b_2) &= \mathcal{O}_{bexp}(b_1)\, \&\, \mathcal{O}_{bexp}(b_2)\\
\mathcal{O}_{bexp}(!(b)) &= !\mathcal{O}_{bexp}(b)
\end{aligned}$$

Statement obfuscation:

$$\begin{aligned}
\mathcal{O}_{stmt}(\texttt{SKIP}) &= \texttt{SKIP}\\
\mathcal{O}_{stmt}(id = a) &= (id = \mathcal{O}_{aexp}(a))\\
\mathcal{O}_{stmt}(s_1\,;\,s_2) &= \mathcal{O}_{aexp}(s_1)\,;\,\mathcal{O}_{aexp}(s_2)\\
\mathcal{O}_{stmt}(\texttt{if }(b)\texttt{ then } s_1 &= \texttt{if }(\mathcal{O}_{bexp}(b))\texttt{ then}\\
\texttt{else } s_2) &\quad\ \mathcal{O}_{stmt}(s_1)\\
&\quad \texttt{else } \mathcal{O}_{stmt}(s_2)\\
\mathcal{O}_{stmt}(\texttt{while }(b)\, s) &= \texttt{while }(\mathcal{O}_{bexp}(b))\\
&\quad\ \mathcal{O}_{stmt}(s)
\end{aligned}$$

value

$\mathcal{O}_{val}$ (value)

Integer encoding

We axiomatize the encoding and decoding of values $O_{val}(v)$ and $D_{val}(v)$.

- Axiom dec_enc_val: $\forall v, D_{val} (O_{val}(v)) = v$.

The memory is obfuscated: notation $O_{mem}(M)$.

- We need a different semantics dedicated to obfuscated programs: a distorted semantics.

See Giacobazzi et. al «Obfuscation by partial evaluation of distorted interpreters», PEPM 2012

Obfuscation seen as a two player game:

- The attacker is an approximate interpreter that is devoted to extract properties of the behavior of a program.
- The defender disguises sensitive properties by distorting code interpretation.

Distorted semantics for integer encoding

$$\vdash M, n :^{\sim} n \qquad \frac{M(x) = \lfloor v \rfloor}{\vdash M, x :^{\sim} v}$$

$$\frac{\vdash M, a_1 :^{\sim} v_1 \qquad \vdash M, a_2 :^{\sim} v_2}{\vdash M, a_1 + a_2 :^{\sim} O_{val}(D_{val}(v_1) + D_{val}(v_2))}$$

- Correctness of expression evaluation
  Lemma integer-encoding-aexp-correct:

  $\forall M,a,v, \quad \vdash M, a : v \quad \Leftrightarrow \quad \vdash O_{mem}(M), O_{aexp}(a) :^{\sim} O_{val}(v)$

Semantics preservation of integer encoding

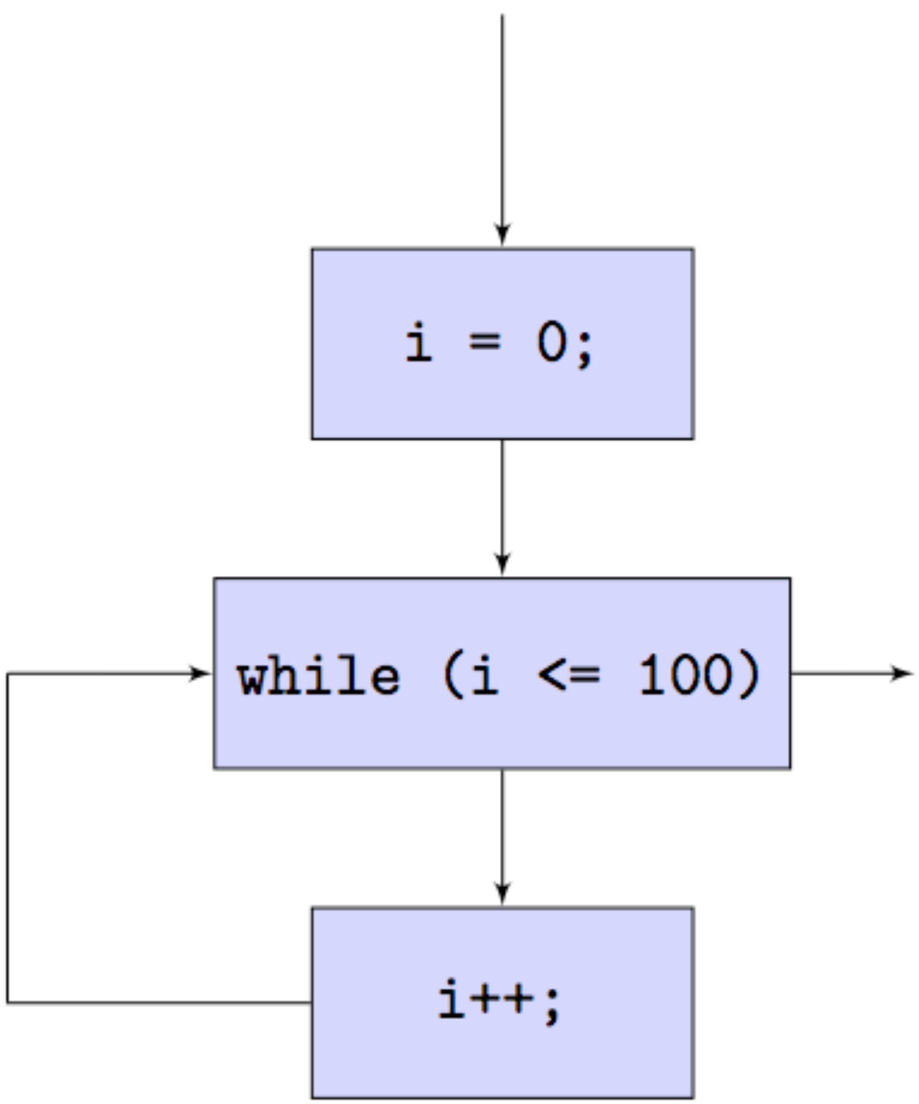
Main properties

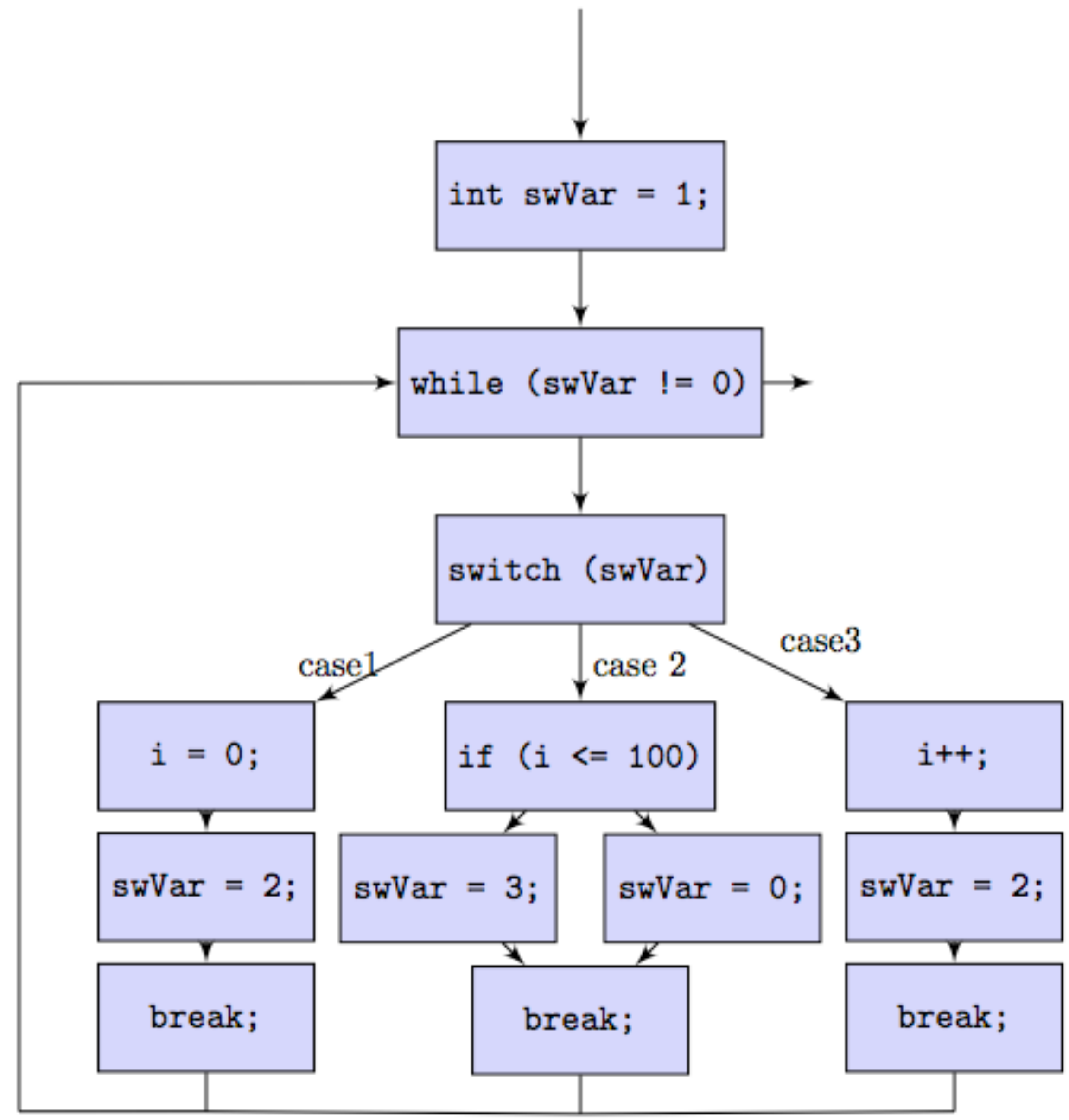
- Lemma obf-aexp-correct:
  $\forall M,a,v, \quad \vdash M, a : v \quad \Leftrightarrow \quad \vdash O_{mem}(M), O_{aexp}(a) \,:\!\tilde{}\; O_{val}(v)$
- Lemma obf-bexp-correct:
  $\forall M,b,v, \quad \vdash M, b : v \quad \Leftrightarrow \quad \vdash O_{mem}(M), O_{bexp}(b) \,:\!\tilde{}\; O_{val}(v)$
- Lemma obf-stmt-correct:
  $\forall M,s, M', \quad \vdash M, s \rightarrow M' \quad \Leftrightarrow \quad \vdash O_{mem}(M), O_{stmt}(s) \rightarrow\!\tilde{}\; O_{mem}(M')$

Intermediate lemmas

- Lemma obf-memory-correct: $\forall M,x,v,\ M(x)= \lfloor v \rfloor \ \Leftrightarrow \ \vdash O_{mem}(M)(x)= \lfloor O_{val}(v) \rfloor$
- Lemma update-obf-correct: $\forall M,x,v, \quad O_{mem}(\ M[x \mapsto v]\ ) = O_{mem}(M)[x \mapsto O_{val}(v)]$
- Lemma update-dob-correct: $\forall M,x,v, \quad D_{mem}(\ M[x \mapsto v]\ ) = D_{mem}(M)[x \mapsto D_{val}(v)]$

Control-flow flattening





Semantics preservation of CFG flattening

We need 4 main intermediate lemmas.

The easiest one is the equivalence between these two loops.

```
while b {
c
}
```

1 execution of c

```
while (n ≤ pc) {
if (pc == n) then
   if (b) then
     pc = m else pc = -1
else c ; pc = n
}
```

2 executions of the loop body

Comparing program obfuscations

- Small imperative language

  Number of intermediate lemmas we wrote in Coq

  Number of PO generated by Why

- Clight language of the CompCert compiler

  Number of (constructors of) inductive predicates

Conclusion

Program obfuscator operating over C programs and integrated in the CompCert compiler

Semantics-preserving code transformation

Intermediate lemmas specify precisely the necessary steps for reverse engineering attacks.

- Opaque predicates = no lemma ! ⇒ straightforward !

The proof measures the difficulty of reverse engineering the obfuscated code.

Questions ?