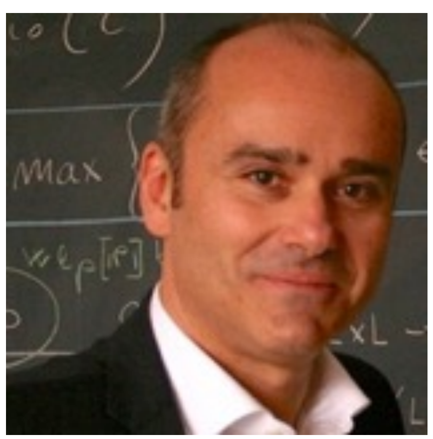
Formal verification of program obfuscations

Sandrine Blazy

joint work with Roberto Giacobazzi and Alix Trieu



IFIP WG 2.11, 2015-11-10

Background: verifying a compiler

Compiler + proof that the compiler does not introduce bugs

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

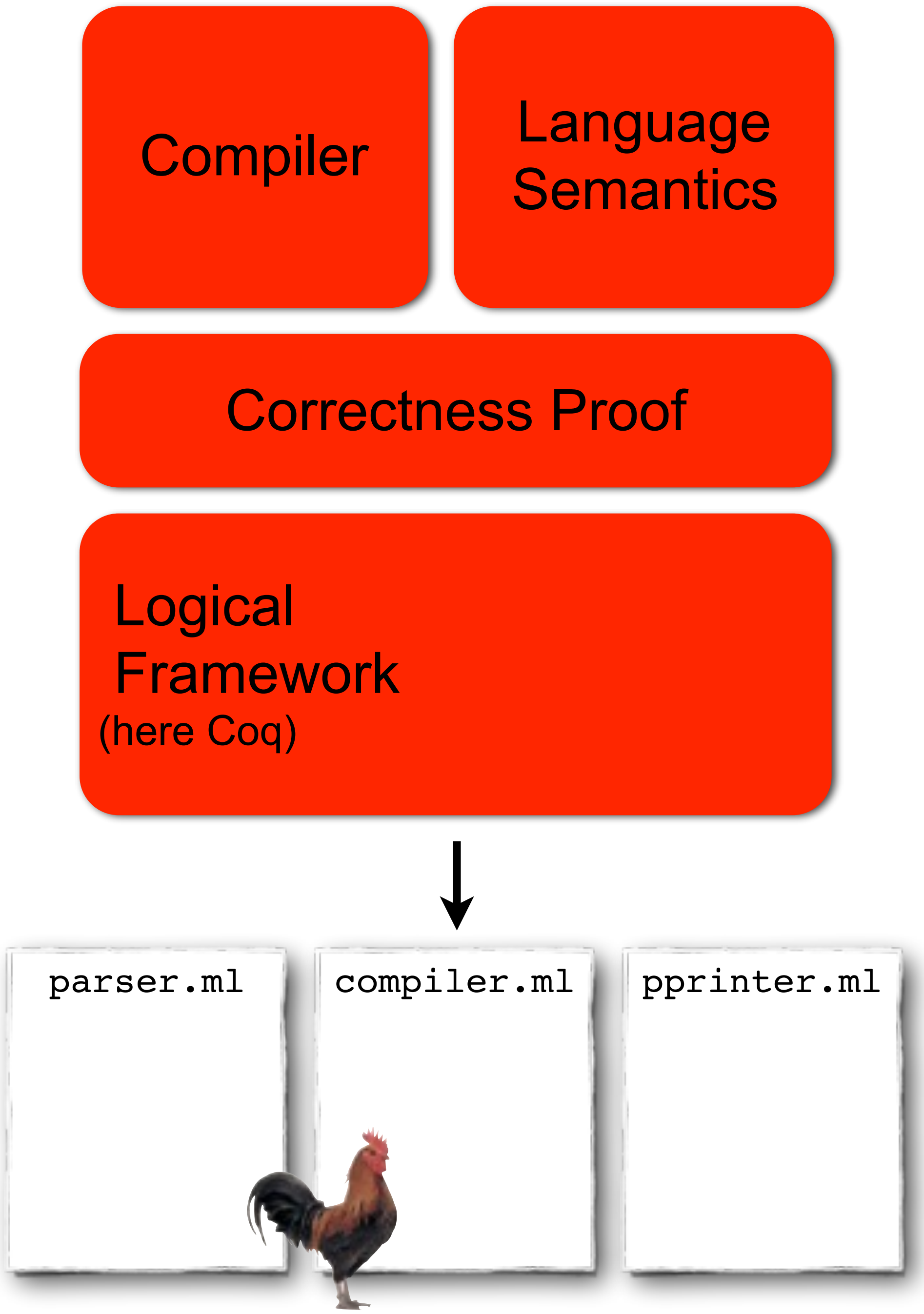
- Fly-by-wire software, Airbus A380 and A400M, FCGU (3600 files):
  mostly control-command code generated from Scade block diagrams + mini. OS

- Commercially available since 2015 (AbsInt company)

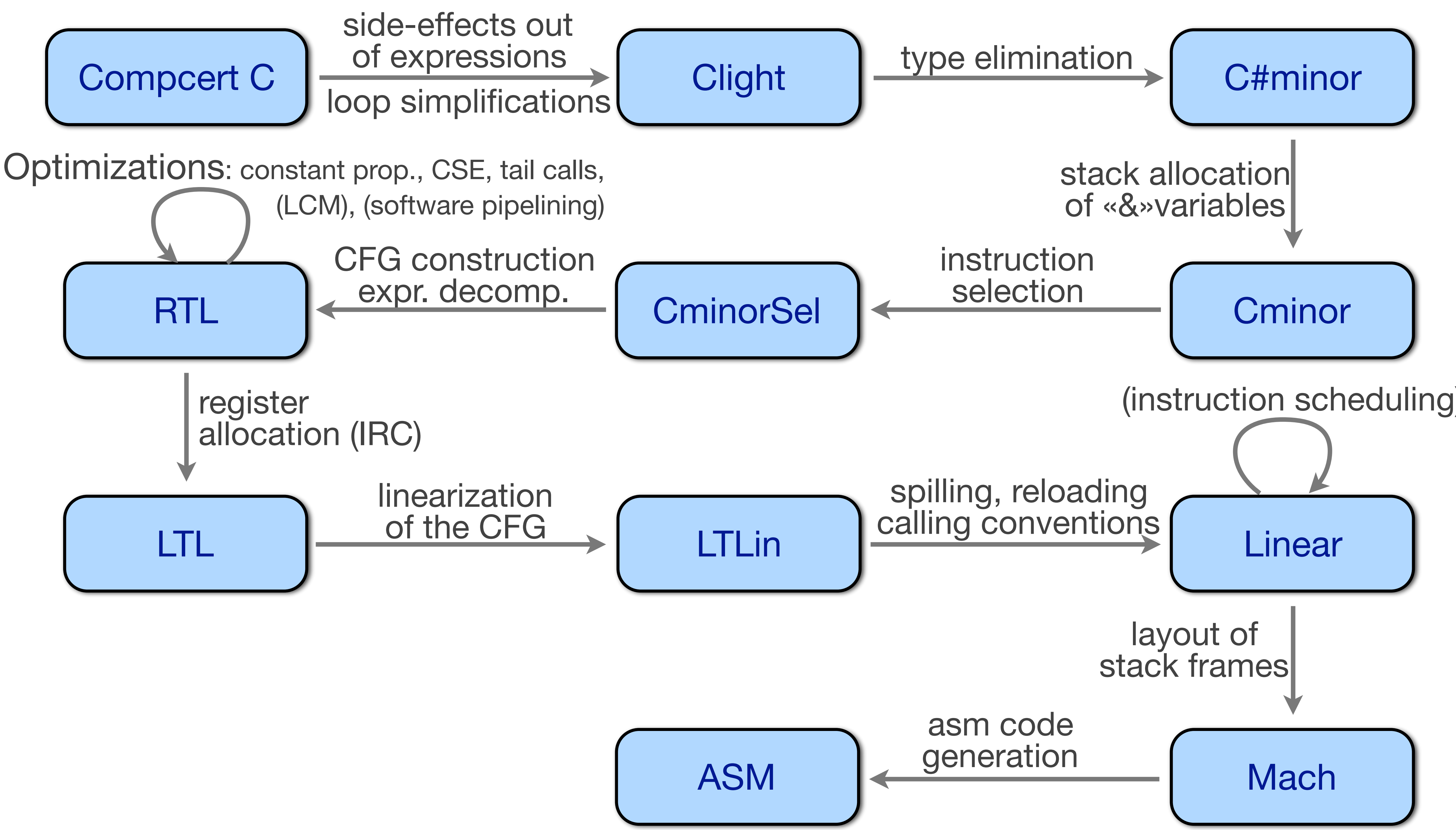
- Formal verification using the Coq proof assistant

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.



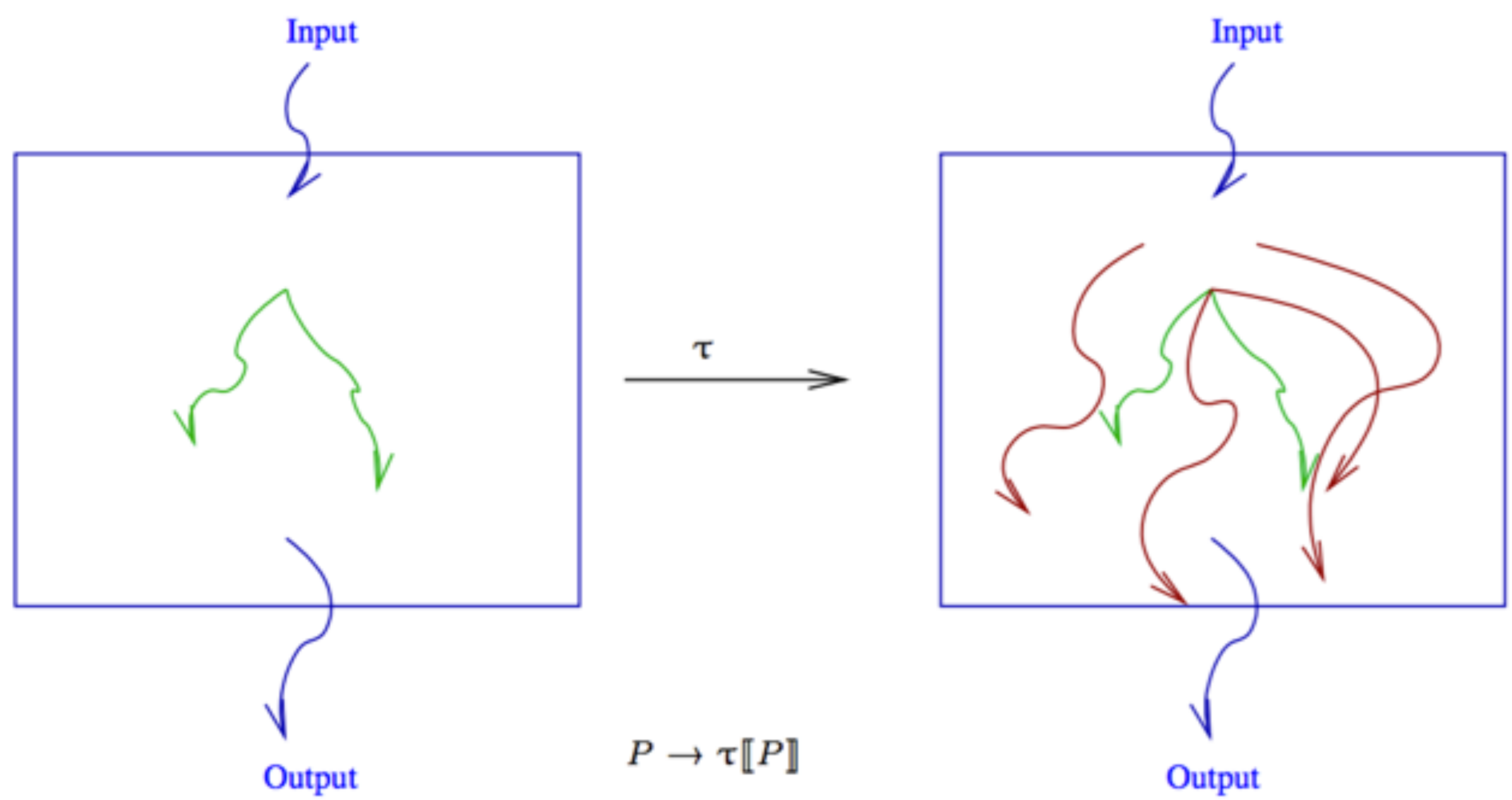
The formally verified part of the compiler



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

and prove that they
preserve the semantics of
Clight 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

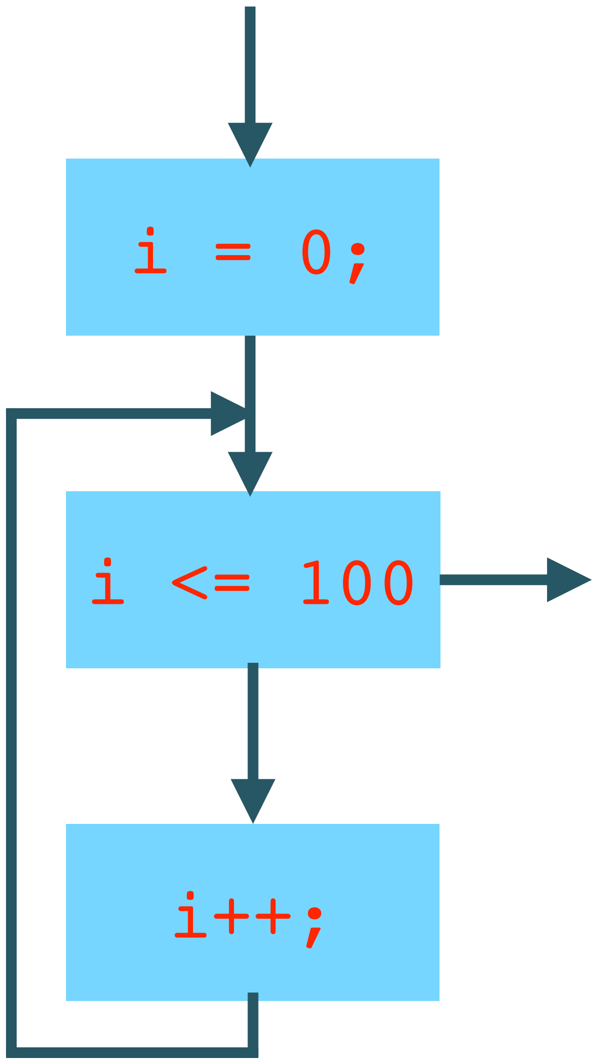
- Hiding data: constant encoding, string encryption, variable encoding,
  variable splitting,
  array splitting, array merging, array folding,
  array flattening

- 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

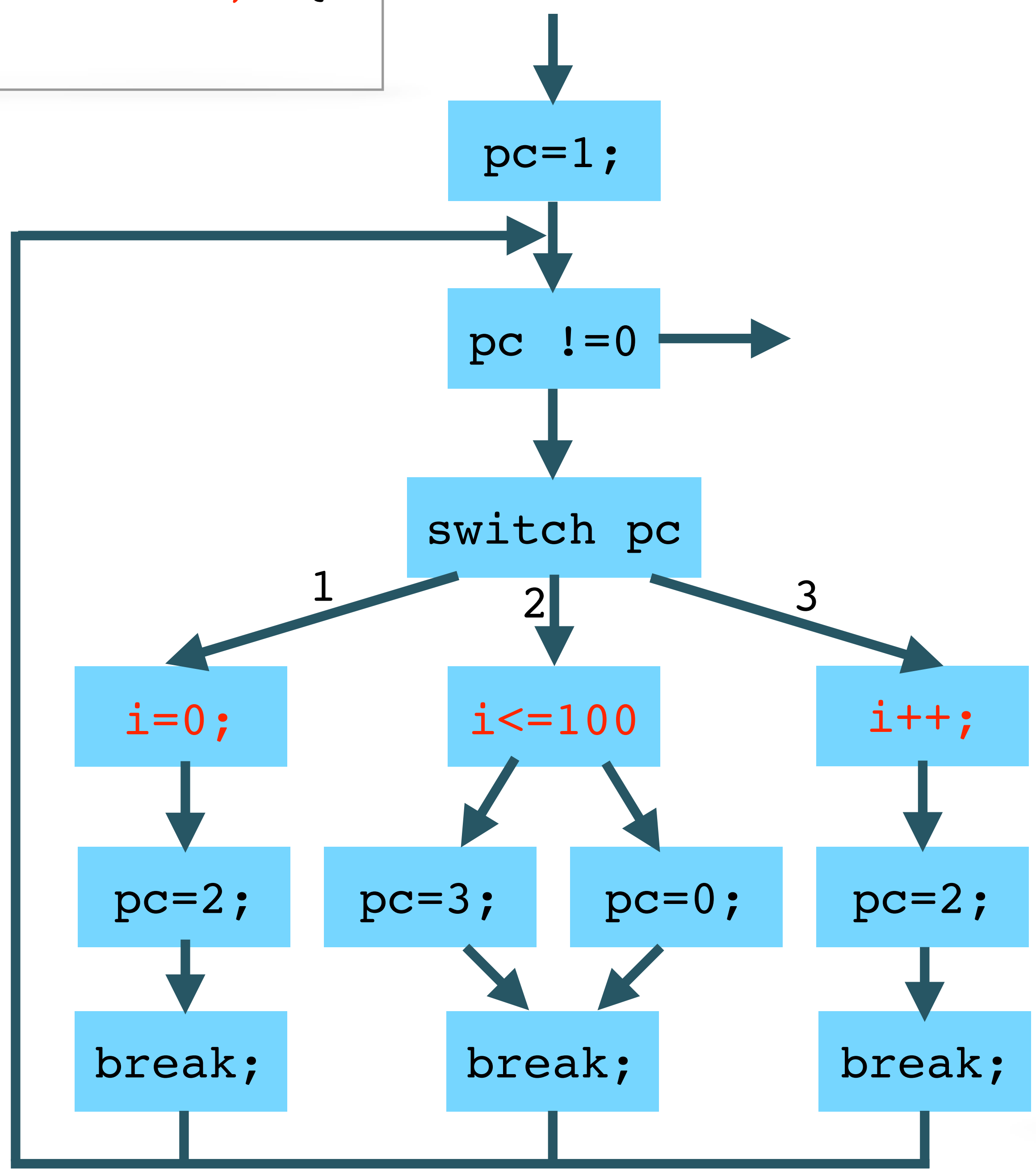



```
i = 0;
while (i <= 100) {
i++; }
```

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

Program obfuscation: control-flow graph flattening

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




```
int pc = 1;
while (pc != 0) {
  switch (pc) {
    case 1 : {
      i = 0;
      pc = 2;
      break; }
    case 2 : {
      if (i <= 100)
        pc = 3;
      else pc = 0;
      break; }
    case 3 : {
      i++;
      pc = 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
control-flow-graph flattening

Clight semantics

Small-step style with continuations, supporting the reasoning on non-terminating programs.

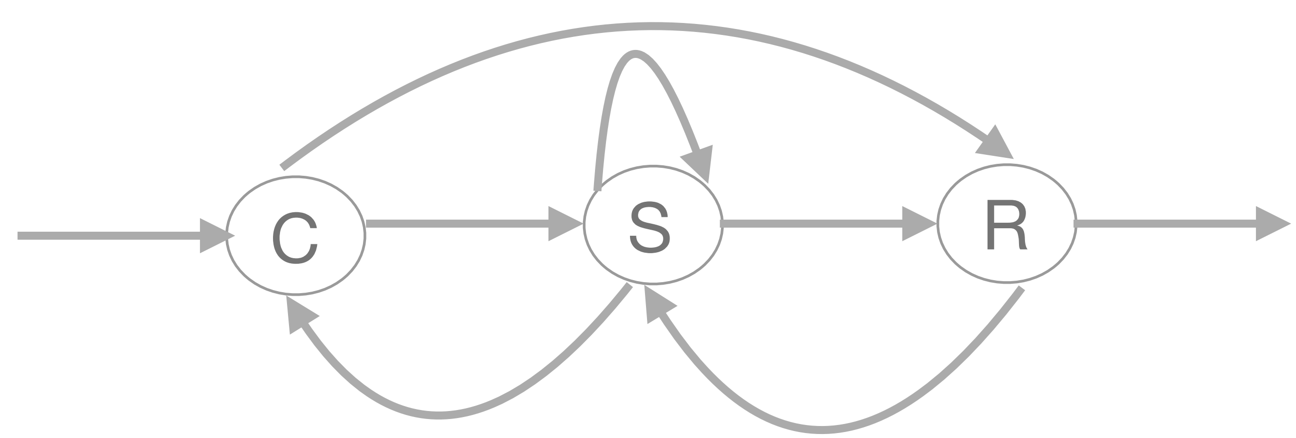
Expressions: 17 rules (big-step)

Statements: 25 rules (small-step)

+ many rules for unary and binary operators, memory loads and stores

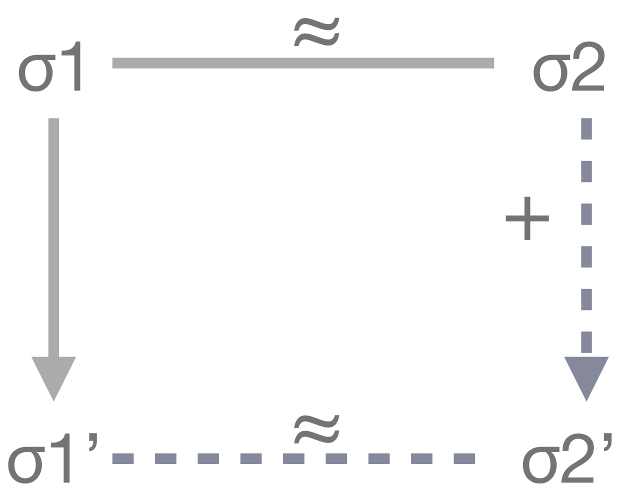
k ::= Kstop | Kseq2 k (* after s1 in s1;s2 *)
| Kloop1 s1 s2 k | Kloop2 s1 s2 k (* after si in (loop s1 s2) *)
| Kswitch k (* catches break statements *)
| Kcall oi f e le k

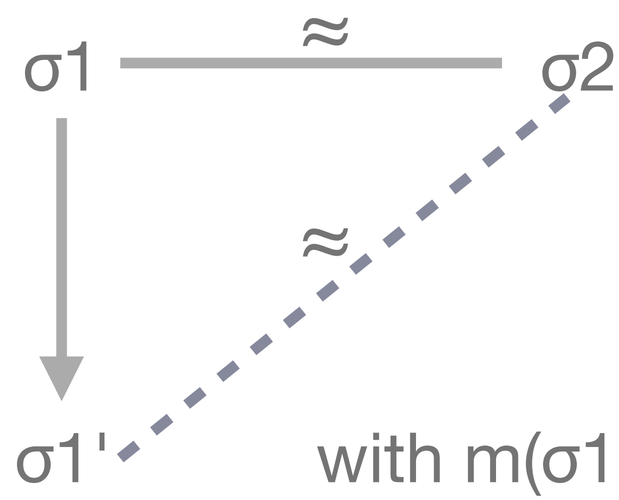
σ ::= C f args k m
| R res k m
| S f s k e le m



(step σ1 σ1’) and also (plus σ2 σ2’)

Correctness of control-flow flattening





with m(σ1’) < m(σ1)

```
step (S f s1;s2 k e le m) (S f s1 (Kseq s2 k) e le m)
```

```
step (S f Skip (Kseq s k) e le m) (S f s k e le m)
```

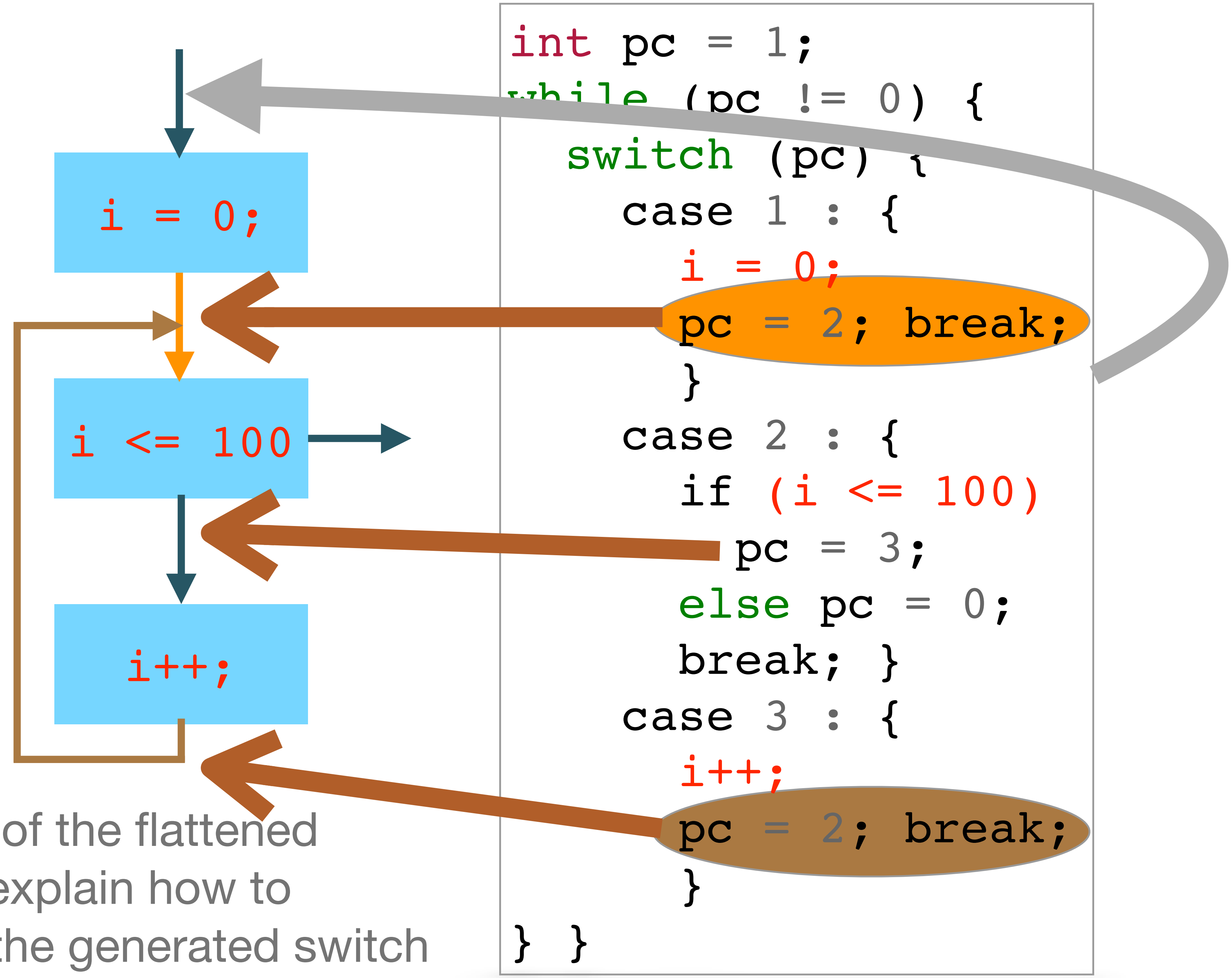
Theorem simulation:

∀ (σ1 σ1':state), step σ1 σ1’ ->

∀ (σ2:state), σ1 ≈ σ2 ->

(∃ σ2', plus σ2 σ2' ⋀ σ1' ≈ σ2') ∨ (m(σ1’) < m(σ1) ∧ σ1' ≈ σ2).

Matching relation between semantic states



Starting from the AST of the flattened program, we need to explain how to rebuild the CFG from the generated switch cases.

Matching relations

$$\frac{\texttt{obf}(\texttt{f}) = \lfloor f' \rfloor \qquad k,k' \in \{\texttt{Kstop},\texttt{Kcall}\} \qquad k \simeq k'}{C(f,a,k,m) \sim C(f',a,k',m)} \quad (1)$$

$$\frac{\texttt{obf}(\texttt{f}) = \lfloor f' \rfloor \qquad k,k' \in \{\texttt{Kstop},\texttt{Kcall}\} \qquad k \simeq k'}{R(v,k,m) \sim R(v,k',m)} \quad (2)$$

$$\frac{\begin{array}{c} \texttt{obf}(\texttt{f}) = \lfloor f' \rfloor \qquad k' \in \{\texttt{Kstop},\texttt{Kcall}\} \\ k \simeq k' \qquad le' = le \dagger \{\texttt{pc}_f \mapsto \lfloor n \rfloor\} \qquad \texttt{flatten}(\texttt{pc}_\texttt{f},\texttt{body}(\texttt{f}),1,0) = \lfloor ls \rfloor \qquad s' = \texttt{while}(\texttt{pc}_f != 0)(\texttt{switch}\,\texttt{pc}_f\,\texttt{ls}) \\ \forall s_1,s_2,k'', \texttt{context}(k) \in \{\texttt{Kloop1}\,s_1 s_2 k'', \texttt{Kloop2}\,s_1 s_2 k''\} \implies \exists n_0 \text{ such that } \texttt{pc}_f, k'' \vdash (\texttt{loop}\,s_1 s_2) \sim ls[n_0] \wedge n_0, \texttt{pc}_f, k \vdash s \approx ls[n] \\ \texttt{context}(k) \in \{\texttt{Kstop},\texttt{Kcall}\} \implies \texttt{pc}_f, k \vdash s \sim ls[n] \end{array}}{S(f,s,k,e,le,m) \sim S(f',s',k',e,le',m)} \quad (3)$$

Figure 8: Matching between states ($\sigma \sim \sigma'$ relation).

$$\frac{\begin{array}{c} ls[n] = (e_1 = e_2; pc = \texttt{next\_stmt};\texttt{break}) \\ \texttt{pc}, k \vdash \texttt{skip} \sim ls[\texttt{next\_stmt}] \end{array}}{pc, k \vdash e_1 = e_2 \sim ls[n]} \quad (4)$$

$$\frac{\begin{array}{c} ls[n] = (\texttt{skip}; pc = \texttt{next\_stmt};\texttt{break}) \\ pc, k \vdash s \sim ls[\texttt{next\_stmt}] \end{array}}{pc, \texttt{Kseq}\,s\,k \vdash \texttt{skip} \sim ls[n]} \quad (5)$$

$$\frac{pc, k \vdash \texttt{s} \sim ls[n]}{pc, \texttt{Kseq}\,s\,k \vdash \texttt{skip} \sim ls[n]} \quad (6)$$

$$\frac{pc, \texttt{Kseq}\,s_2\,k \vdash s_1 \sim ls[n]}{pc, k \vdash \texttt{s}_1;\texttt{s}_2 \sim ls[n]} \quad (7)$$

$$\frac{\begin{array}{c} ls[n] = (\texttt{if}\ b\ \texttt{then}\ pc = n+1\ \texttt{else}\ pc = n+1+|\texttt{s}_1|;\texttt{break}) \\ pc, k \vdash \texttt{s}_1 \sim ls[n+1] \qquad pc, k \vdash \texttt{s}_2 \sim ls[n+1+|\texttt{s}_1|] \end{array}}{pc, k \vdash \texttt{if}\ b\ \texttt{then}\ \texttt{s}_1\ \texttt{else}\ \texttt{s}_2 \sim ls[n]} \quad (8)$$

$$\frac{\begin{array}{c} ls[n] = (pc = n+1;\texttt{break}) \\ n, pc, \texttt{Kloop1}\,s_1\,s_2\,k \vdash s_1 \approx ls[n+1] \end{array}}{pc, k \vdash \texttt{loop}\,\texttt{s}_1\,\texttt{s}_2 \sim ls[n]} \quad (9)$$

Figure 9: Matching between statements ($\texttt{pc}, k \vdash s \sim ls[n]$ relation).

$$\frac{\begin{array}{c} ls[n] = (e_1 = e_2; pc = \texttt{next\_stmt};\texttt{break}) \\ n_0, \texttt{pc}, k \vdash \texttt{skip} \approx ls[\texttt{next\_stmt}] \end{array}}{n_0, pc, k \vdash e_1 = e_2 \approx ls[n]} \quad (4')$$

… (9')

$$\frac{\begin{array}{c} ls[n] = (\texttt{skip}; pc = \texttt{next\_stmt};\texttt{break}) \\ n_0, pc, \texttt{Kloop2}\,s_1\,s_2\,k \vdash s_2 \approx ls[\texttt{next\_stmt}] \end{array}}{n_0, pc, \texttt{Kloop1}\,s_1\,s_2\,k \vdash \texttt{skip} \approx ls[n]} \quad (10)$$

$$\frac{n_0, pc, \texttt{Kloop2}\,s_1\,s_2\,k \vdash s_2 \approx ls[n]}{n_0, pc, \texttt{Kloop1}\,s_1\,s_2\,k \vdash \texttt{skip} \approx ls[n]} \quad (11)$$

$$\frac{ls[n] = (\texttt{skip}; pc = n_0;\texttt{break})}{n_0, pc, \texttt{Kloop2}\,s_1\,s_2\,k \vdash \texttt{skip} \approx ls[n]} \quad (12)$$

$$\frac{}{n_0, pc, \texttt{Kloop2}\,s_1\,s_2\,k \vdash \texttt{skip} \approx ls[n_0]} \quad (13)$$

$$\frac{\begin{array}{c} ls[n] = (pc = \texttt{next\_stmt};\texttt{break}) \\ pc, k \vdash \texttt{skip} \sim ls[\texttt{next\_stmt}] \end{array}}{n_0, pc, \texttt{Kloop2}\,s_1\,s_2\,k \vdash \texttt{break} \approx ls[n]} \quad (14)$$

Figure 10: Matching between statements ($n_0, \texttt{pc}, k \vdash s \approx ls[n]$ relation).

Implementation and experiments

1200 lines of spec + 4250 lines of proofs + reused CompCert libraries

The comparison with Obfuscator-LLVM revealed a slowdown in the execution of our obfuscated programs, due to a number of skip statements that are generated by the first pass of CompCert.

- Trick to facilitate the proof: use skip statements to materialize evaluation steps of non-deterministic expressions.

Solution: add a pass that eliminates skip statements in skip;s sequences

Experimental results

		CompCert	no skips + obf.		Obfuscator-LLVM		Ratio
Program (1)	LoC (2)	Original (3)	Obfuscated (4)	Ratio (5)	Obfuscated (6)	Ratio (7)	LLVM / no skips + obf. (8)
aes.c	1453	1.015	3.256	3.207	2.290	2.256	0.703
almabench.c	351	0.452	0.781	1.727	0.600	1.327	0.768
binarytrees.c	164	5.001	6.007	1.201	5.387	1.077	0.896
bisect.c	377	4.675	10.127	2.166	24.893	5.324	2.457
chomp.c	368	1.393	4.308	3.092	4.654	3.340	1.080
fannkuch.c	154	0.265	3.306	12.475	6.504	24.543	1.967
fft.c	191	0.095	0.161	1.694	0.302	3.178	1.876
fftsp.c	196	0.001	0.002	2.000	0.004	4.000	2.000
fftw.c	89	2.059	17.817	8.653	6.419	3.117	0.360
fib.c	19	0.164	0.395	2.408	0.662	4.036	1.676
integr.c	32	0.052	0.167	3.211	0.148	2.846	0.886
knucleotide.c	369	0.080	0.152	1.900	0.138	1.725	0.907
lists.c	81	0.386	5.047	13.075	1.214	3.145	0.240
mandelbrot.c	92	1.302	5.559	4.269	24.173	18.566	4.349
nbody.c	174	4.981	17.062	3.425	17.489	3.511	1.025
nsieve.c	57	0.113	0.548	4.849	1.497	13.247	2.731
nsievebits.c	76	0.101	0.389	3.851	0.929	9.198	2.388
perlin.c	75	8.241	32.876	3.989	53.520	6.494	1.627
qsort.c	50	0.293	1.342	4.580	2.703	9.225	2.014
sha3.c	2233	5.202	34.601	6.651	8.321	1.599	0.240

Conclusion

Competitive program obfuscator operating over C programs, integrated in the CompCert compiler

Semantics-preserving code transformation

Future work

- Combine CFG flattening with other simple obfuscations
- The proof measures the difficulty of reverse engineering the obfuscated code.
    - Study how to count the size of lambda-terms
    - Semantics of proofs as independent objects (focused proof systems)

Questions ?