Verified translation validation of static analyses

Sandrine Blazy

joint work with Gilles Barthe, Vincent Laporte, David Pichardie and Alix Trieu

IFIP WG 1.9/2.15, Leuven, 2017-05-11

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

We prove the following semantic preservation property:

> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> and S has a safe behavior,
> then «C behaves like S».

Behaviors = termination / divergence / undefined («going wrong»)
+ trace of I/O operations performed

Our methodology

We program the compiler inside Coq.

```
Definition compiler (S: program) := ...
```

We state its correctness w.r.t. a formal specification of the language semantics.

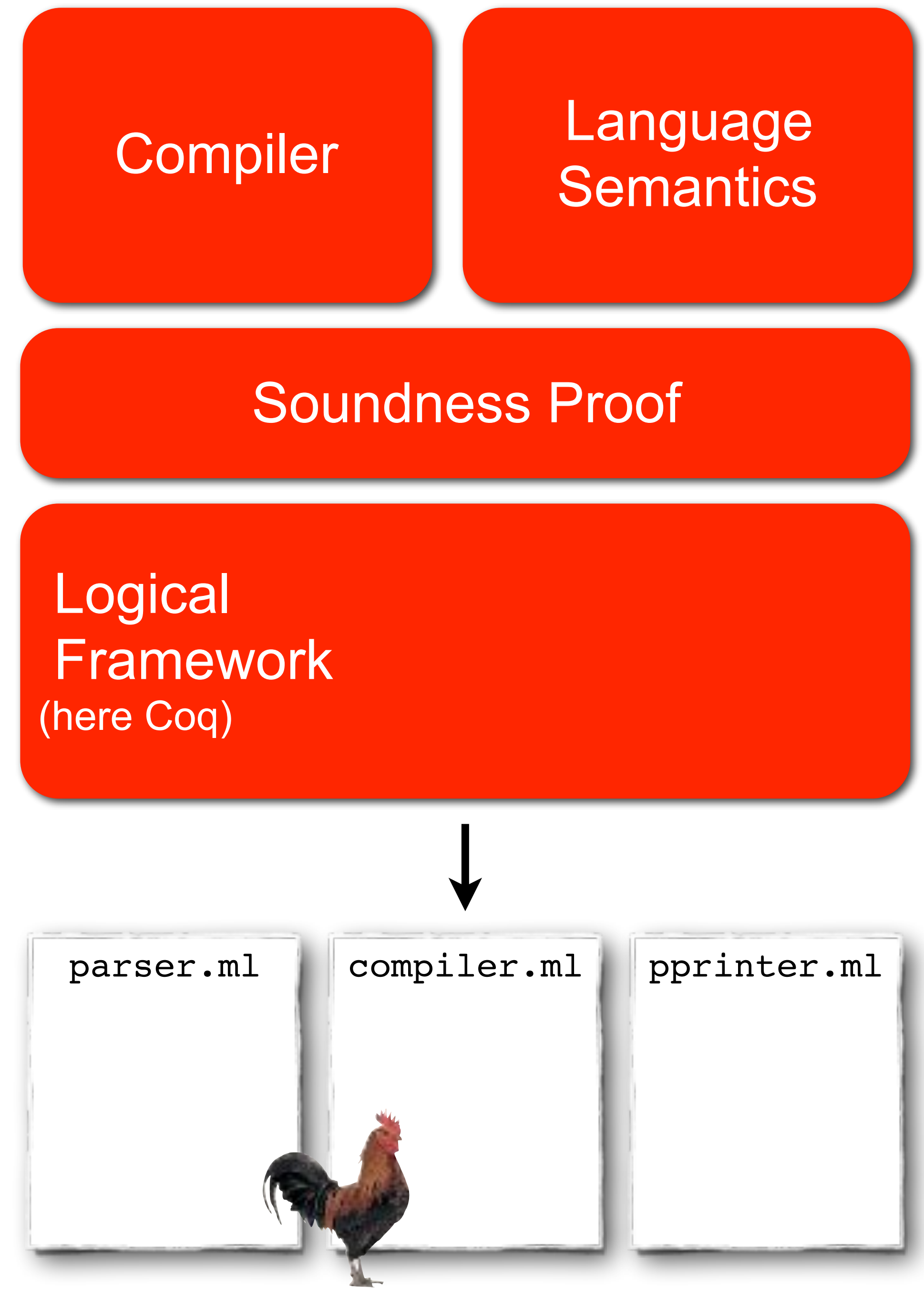
```
Theorem compiler_is_correct :
∀ S C, compiler S = OK (C) → safe (S) →
      «C behaves like S».
```

We interactively and mechanically prove this theorem

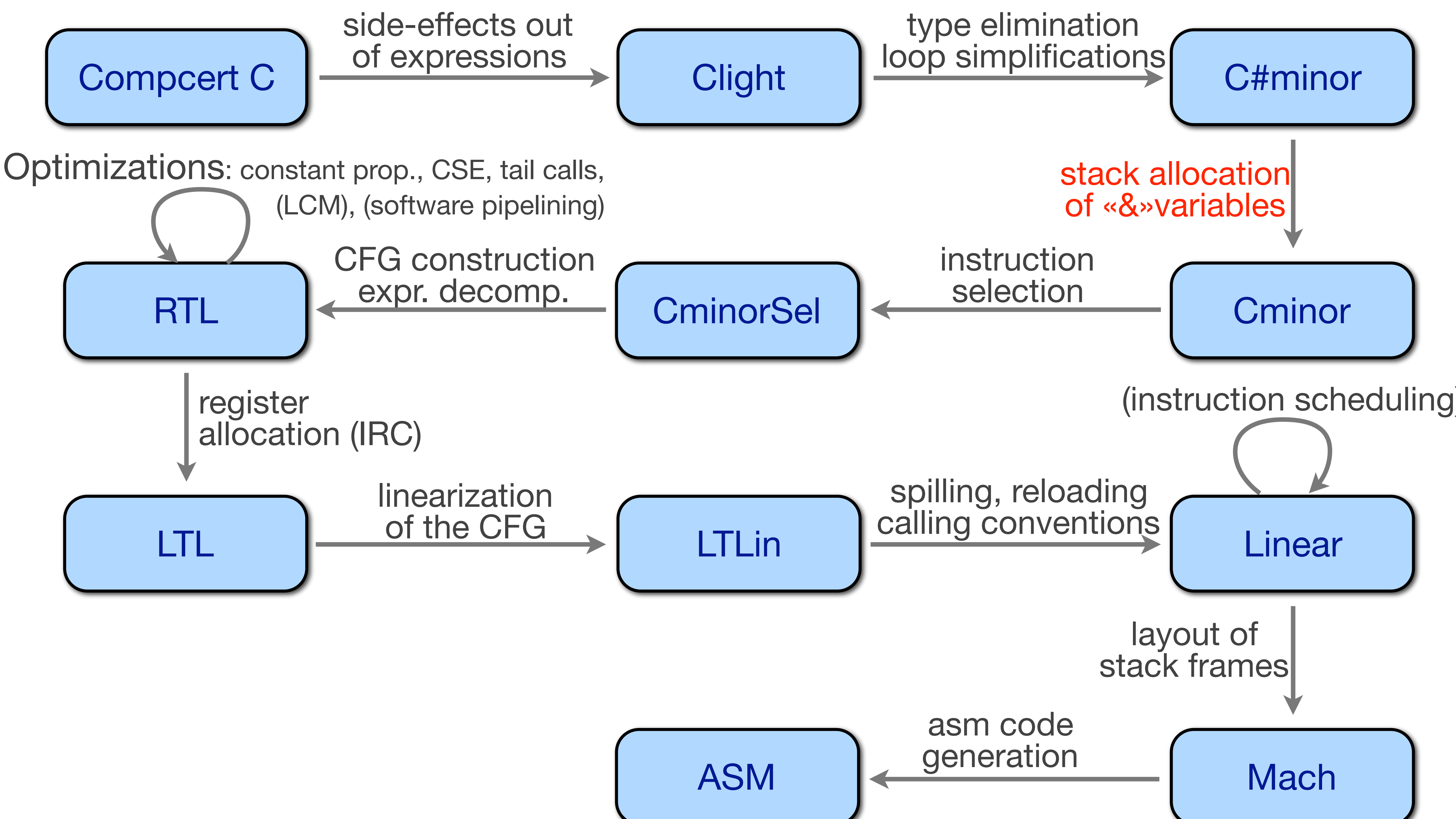
```
Proof. ...(* a few months later *) ...
Qed.
```

We extract an OCaml implementation of the compiler.

```
Extraction compiler.
```



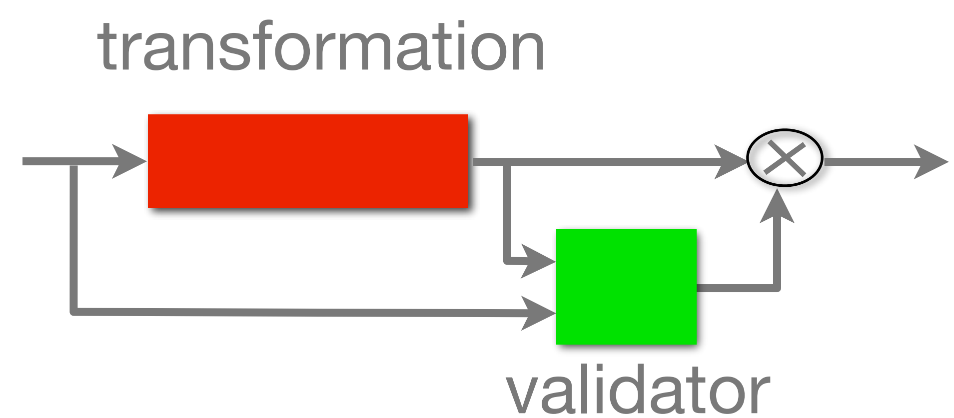
The formally verified part of the CompCert

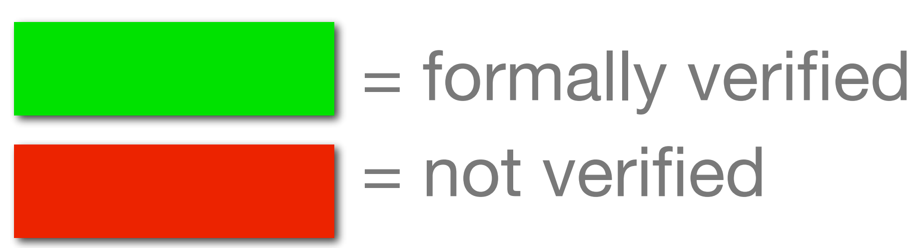


Verification patterns (for each compilation pass)

Verified transformation

Verified translation validation





Same methodology

We program the static analyzer inside Coq.

```
Definition analyzer (p: program) := ...
```

We state its correctness w.r.t. a formal specification of the language semantics.

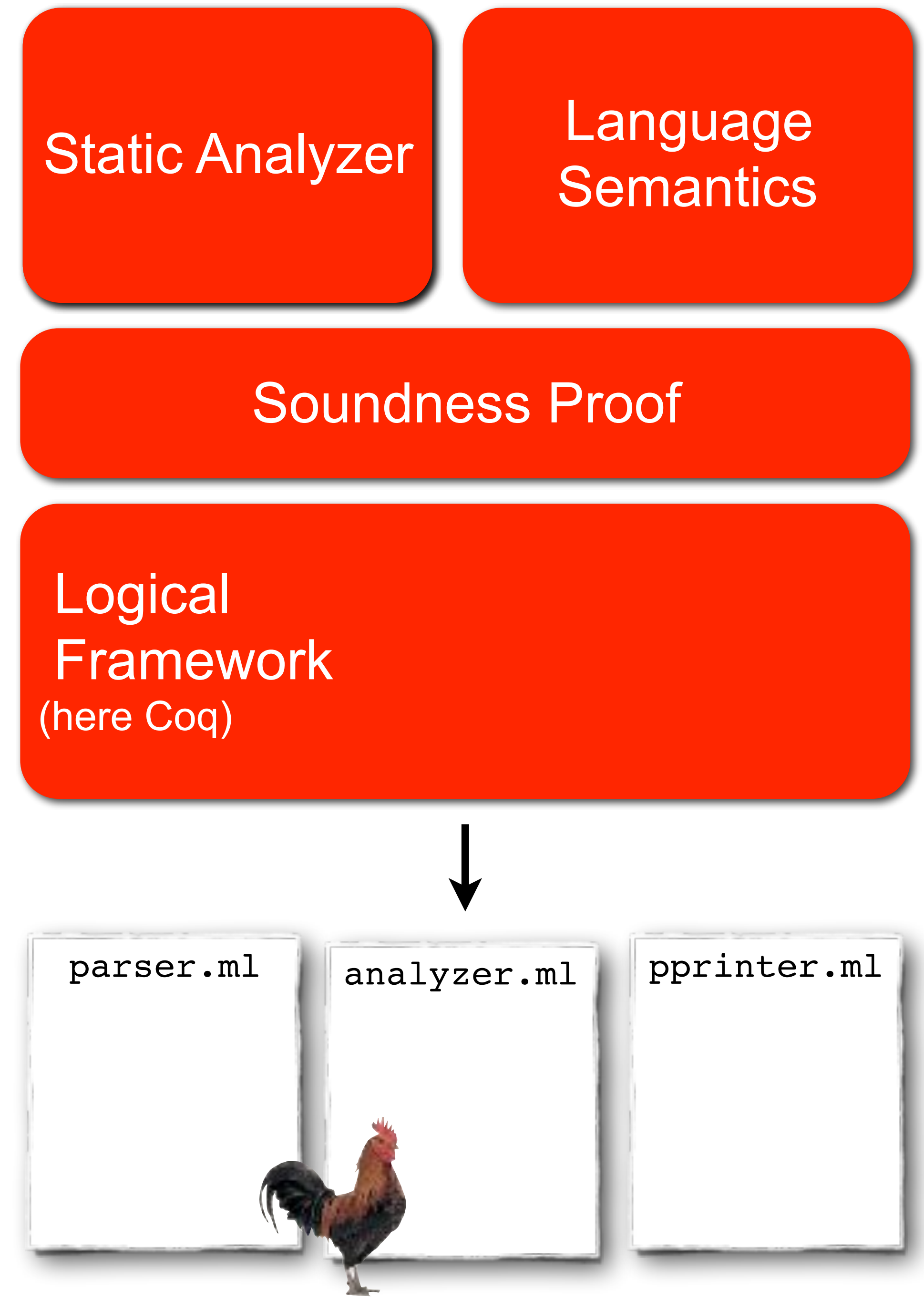
```
Theorem analyzer_is_sound :
 ∀ P, analyzer P = Yes →
      safe(P).
```

We interactively and mechanically prove this theorem

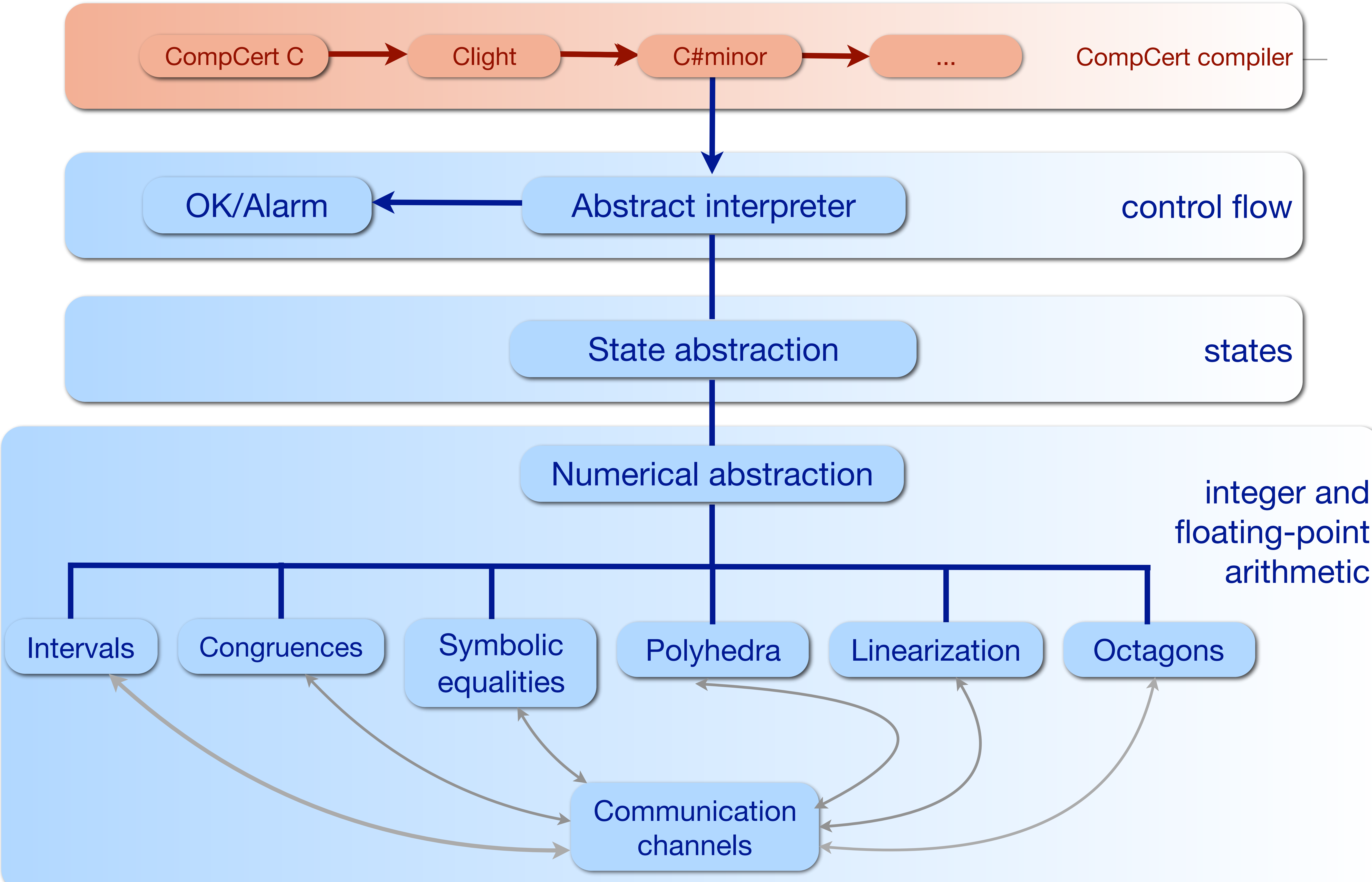
```
Proof. ... (* a few months later *) ...
Qed.
```

We extract an OCaml implementation of the analyzer.

```
Extraction analyzer.
```



The Verasco static analyzer



Abstract interpretation of low-level programs ?

- Abstract interpretation traditionally performed at source level

- Need for analyzing lower-levels
  - Ex1: compiler optimization (intermediate level)
  - Ex2: security analysis performed at assembly level
  - Difficulty of the analysis (e.g. keeping track of symbolic equalities between values contained in memory cells - incl. points-to information - and alignment of memory accesses)

- Our solution: a general and lightweight methodology for carrying the results of a source analyzer down to lower-level representations
  - 3 use cases: CSE optimization, constant-time analysis, resource analysis

Our methodology

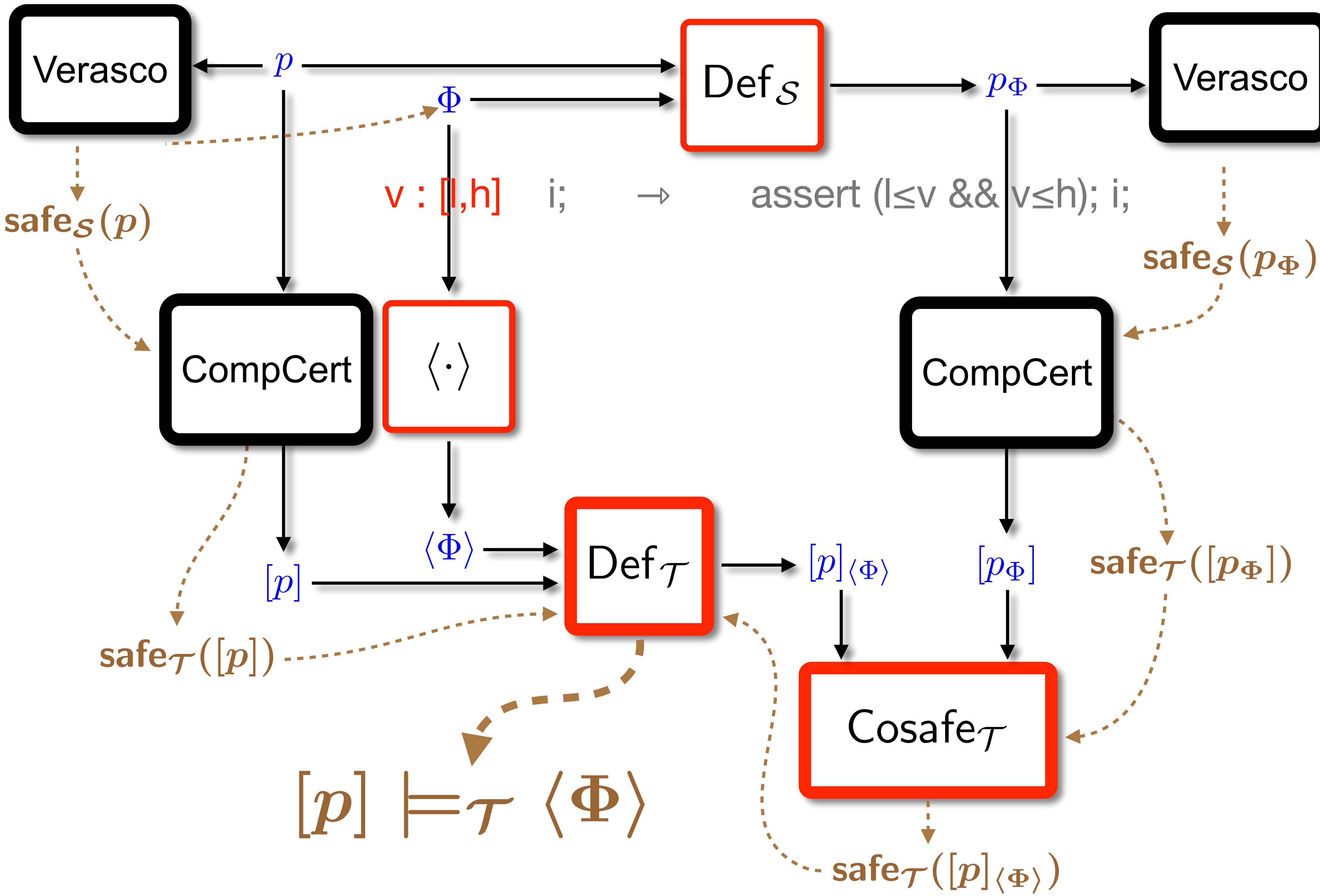
- Inlining enforceable properties
  - properties that can be enforced using runtime monitors
    Inlining a monitor yields a defensive form (i.e. a program instrumented with runtime checks)
    Enforcing a program to follow a property amounts to checking that it is safe.

```
int *x;
int t[3];
/* … */
y = *x;
```

```
int *x;
int t[3];
/* … */
assert (x==t || x==t+1 || x==t+2);
y = *x;
```

- Relative safety: P_1 is safe under the knowledge that P_2 is safe
  - An instance of relational verification

Methodology



Instantiation of the methodology

Focus on points-to annotations

Each memory access is annotated with an optional set of symbolic pointers.

```
/* x → t1[2..4] ∪ t2[6..8] */
assert (x==t1+2||x==t1+3||x==t1+4 ||x==t2+6||x==t2+7||x==t2+8);
y = *x;
```

Difficulty: handling local variables

```
int main(void) { int t_1[12], t_2[9001];
  ... call to f ...  return …}

int f(int* z) { int y, *x;
/* ... */
/* x → main@t_1[2..4] ∪ main@t_2[6..8] */
y = *x;
/* ... */ return …}
```

Forging pointers: the shadow stack

```
int f(int* z) { int y, *x;
/* ... */
/* x → main@t_1[2..4] ∪ main@t_2[6..8] */
y = *x;
/* ... */ return …}
```

Difficulty: handling local variables
Solution: use of a shadow stack

- We need to compute some concrete pointers that are symbolically given by the annotations.
- We make each function leak a pointer to its stack frame into a global variable (a.k.a. the shadow stack).

Example of shadow stack

```
int* STK[2048];                       shadow stack
int CNT = 0;                          stack pointer

int main(void) {
  int main_stk[9013];
  CNT = CNT+1;                        prologue (push)
  STK[CNT] = main_stk;
  /* ... call to f ... */
  CNT = CNT-1; return … }             epilogue (pop)

int f(int* z) {
  int f_stk[2];
  CNT = CNT+1;
  STK[CNT] = f_stk;
  /* ... */
  /* x → -1[2..4] ∪ -1[18..20] */
  assert(f_stk[1]==STK[CNT-1]+2 || f_stk[1]==STK[CNT-1]+3 || ... );
  f_stk[0] = *(f_stk[1]);
  /* ... */
  CNT = CNT-1; return …}
```

Use case: cryptographic constant-time

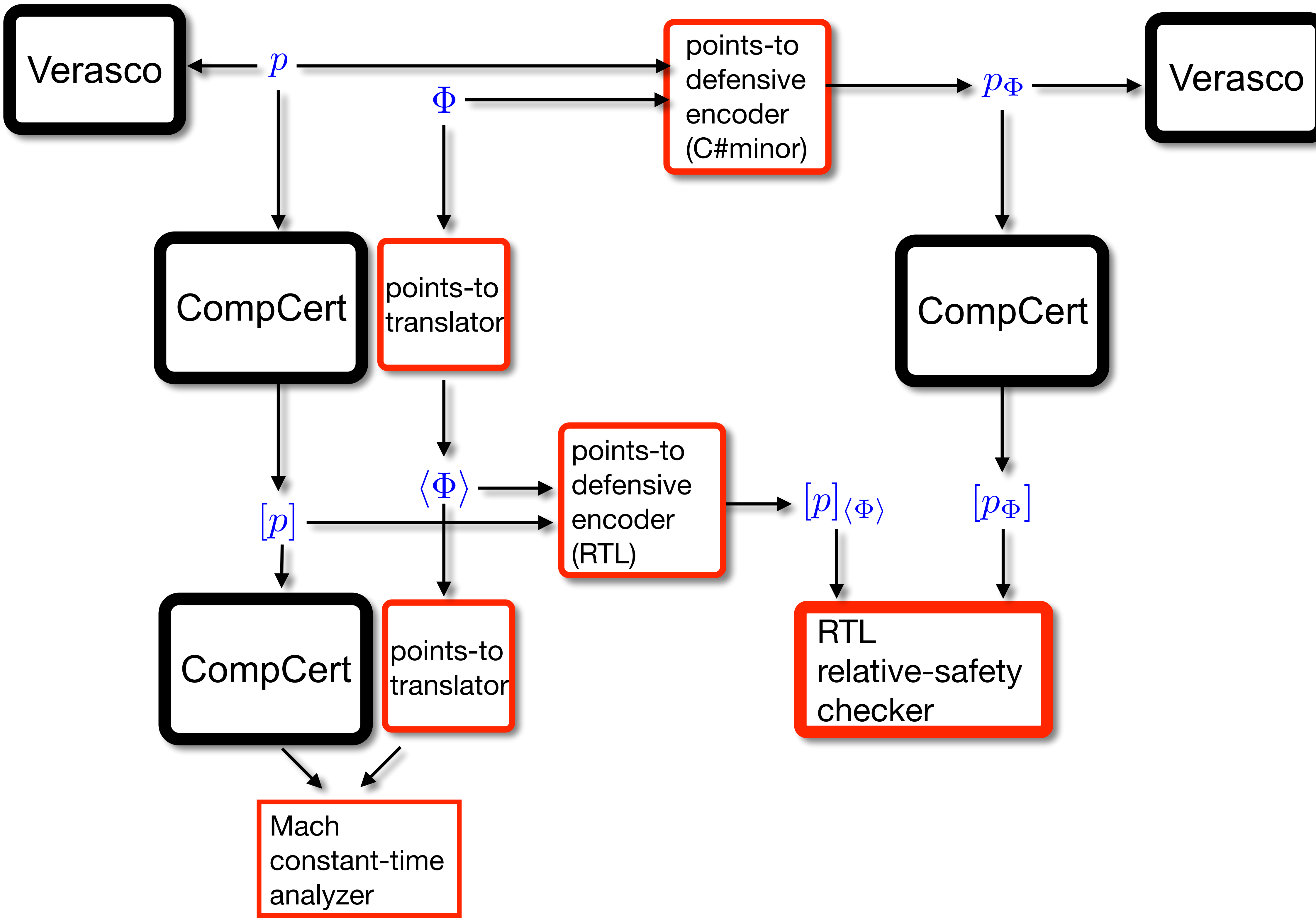
Constant-time policy: the control flow and sequence of memory accesses of a program do not depend on some of its inputs (tagged as secret).

Use of the points-to information from Verasco to keep track of security levels, and exploit this information in an information-flow type system (Mach level)

- avoid the need to rewrite programs
- handle larger programs

We were able to automatically prove that programs verify the constant-time policy.
Benchmarks: mainly PolarSSL and NaCl cryptographic libraries

Use case: cryptographic constant-time



Conclusion

Lightweight approach to formally verify translation of static analysis results (lowering of points-to annotations) in a formally verified compiler

Two main ingredients: inlining enforceable properties and differential verification

Improves a previous security analysis at pre-assembly level

Future work

Improve Verasco to perform a very precise taint analysis

- Relies on a tainted semantics
- Encouraging results on a representative benchmark
- Main theorem: any safe program w.r.t. the tainted semantics is constant time (paper proof)

Add obfuscation transformations and check that they do not introduce side-channels

References

- G. Barthe, S. Blazy, V. Laporte, D. Pichardie, A. Trieu. **Verified translation validation of static analyses**. Computer Security Foundations Symposium (CSF), 2017.
- S. Blazy, V. Laporte, D. Pichardie. **An abstract memory functor for verified C static analyzers**. ICFP 2016.
- J.H. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie. **A formally-verified static analyzer**. POPL 2015.
- G. Barthe, G.Bertate, J.D.Campo, C.Luna, D. Pichardie. **System-level non-interference for constant-time cryptography**. Conference on Computer and Communications Security (CCS), 2014.
- F. Schneider. **Enforceable security policies**. ACM Transactions on Information and System Security. 2000.
- M. Dam, B. Jacobs, A. Lundblad, F. Piessens. **Provably correct inline monitoring for multithreaded Java-like programs**. Journal of Computer Security, 2010.

Questions ?