Verified translation validation of static analyses

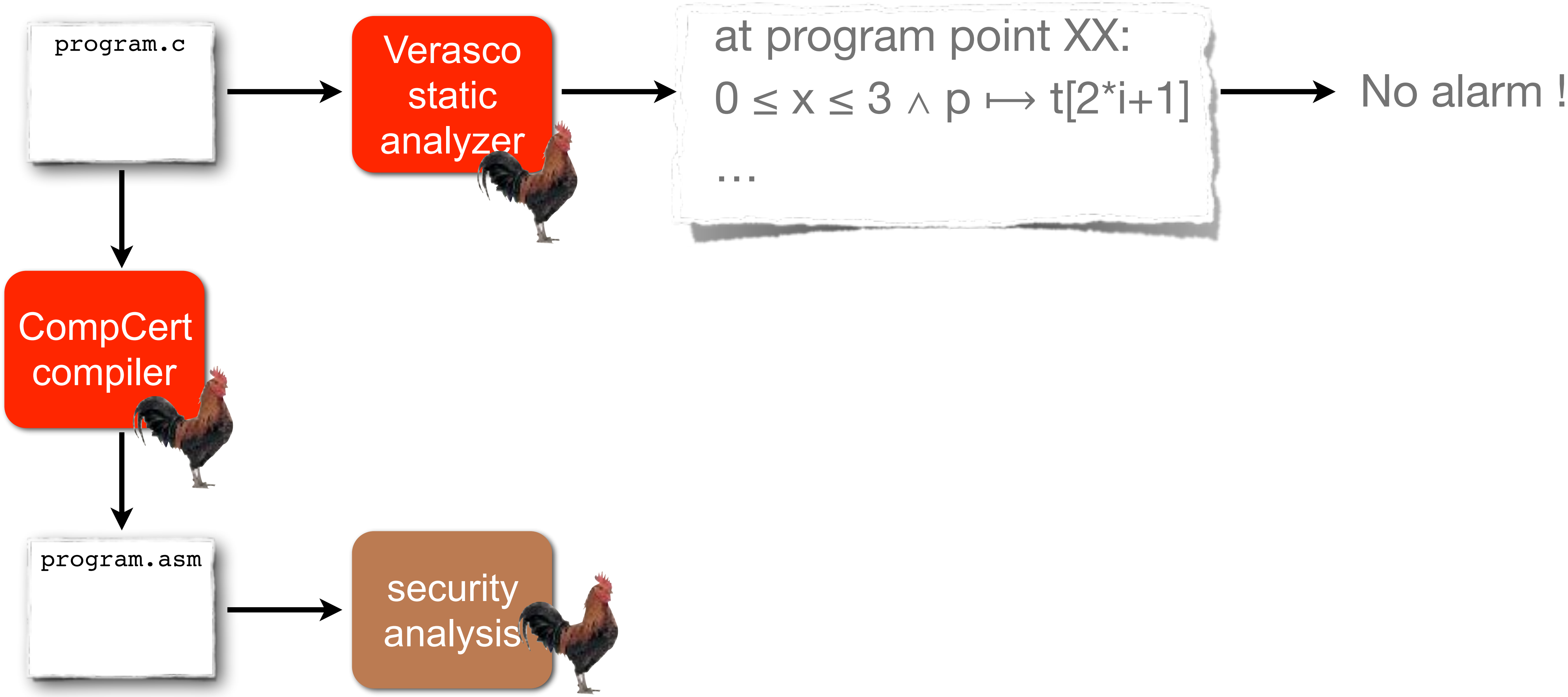
Sandrine Blazy
Univ. Rennes, CNRS IRISA, Inria

---

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

IFIP WG 2.11 京都大学 2018年6月8日

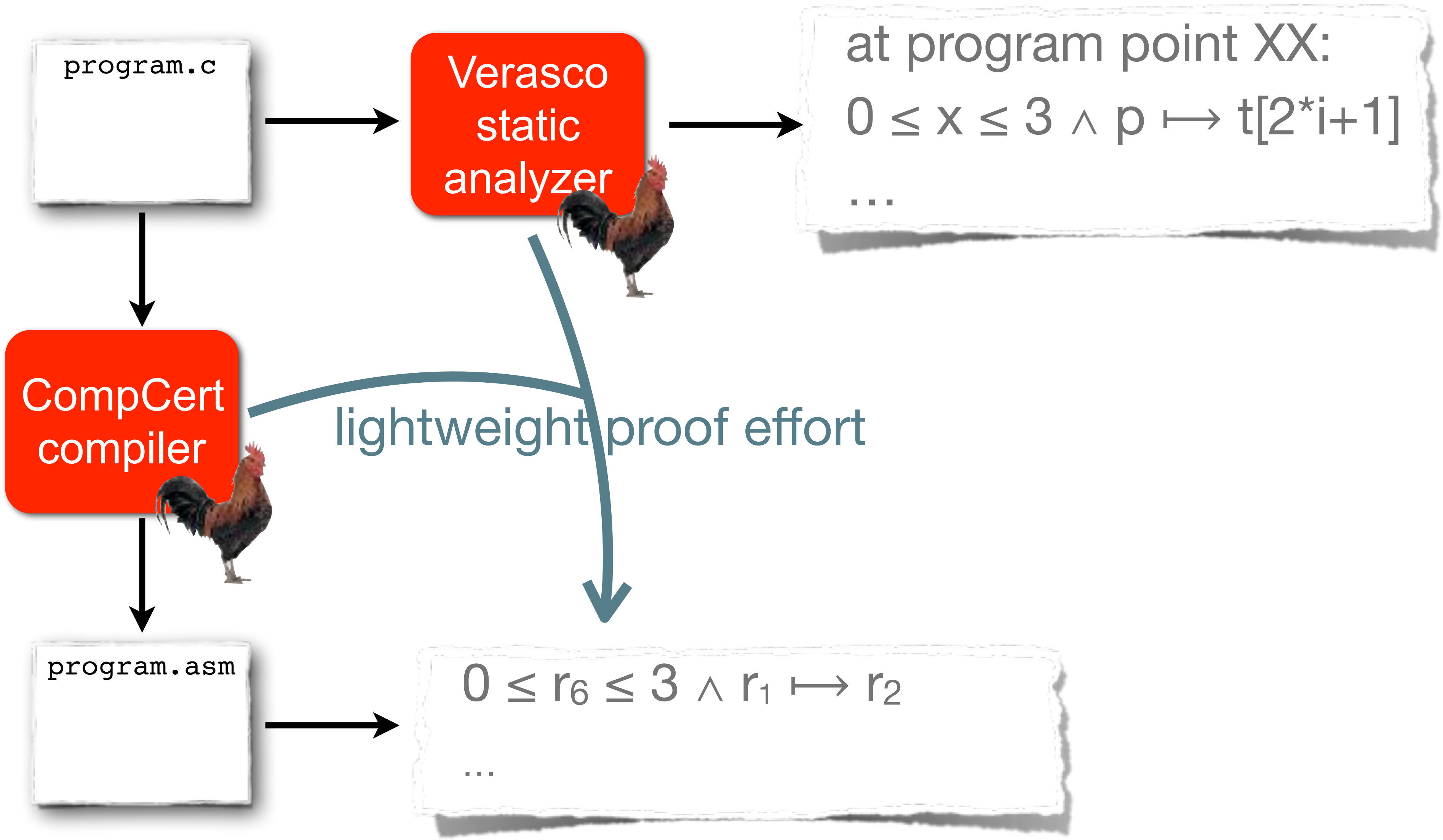
How to check security properties of implementations ?



Cryptographic constant-time property :

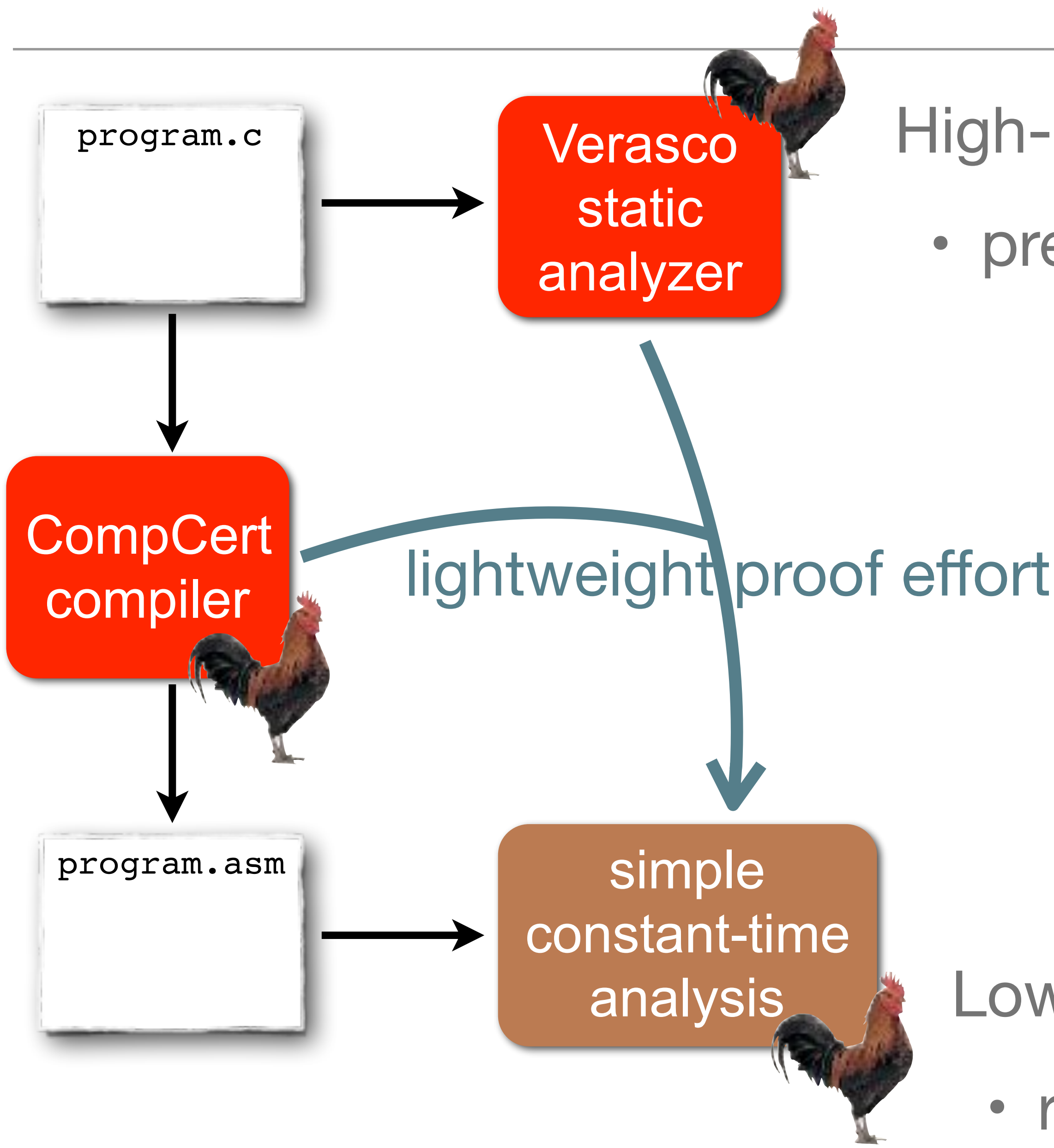
- branching do not depend on secrets
- memory accesses do not depend on secrets

This talk



General and lightweight methodology for carrying the results of a source analyzer down to lower-level representations

This talk



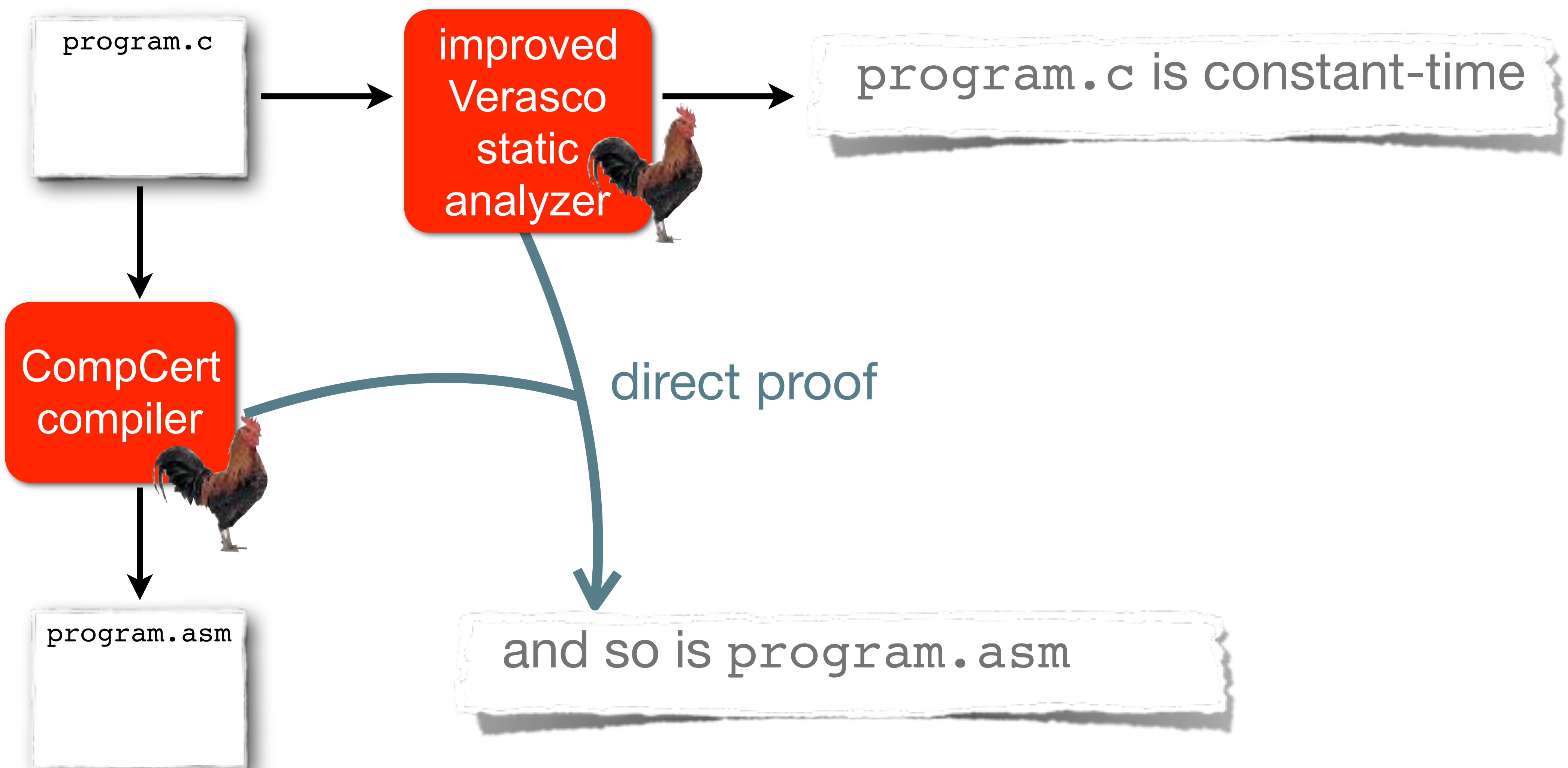
High-level abstract interpreter :

- precise (e.g. precise points-to analysis)

Low-level data-flow analysis :

- requires to manually rewrite programs
- lack of precision
- does not scale

Alternative solution (work in progress)



Brittle approach : it requires to

- instrument all the semantics of CompCert
- adapt and redo all the proofs
- define new proof principles

Background : the formally verified tools CompCert and Verasco

CompCert 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.

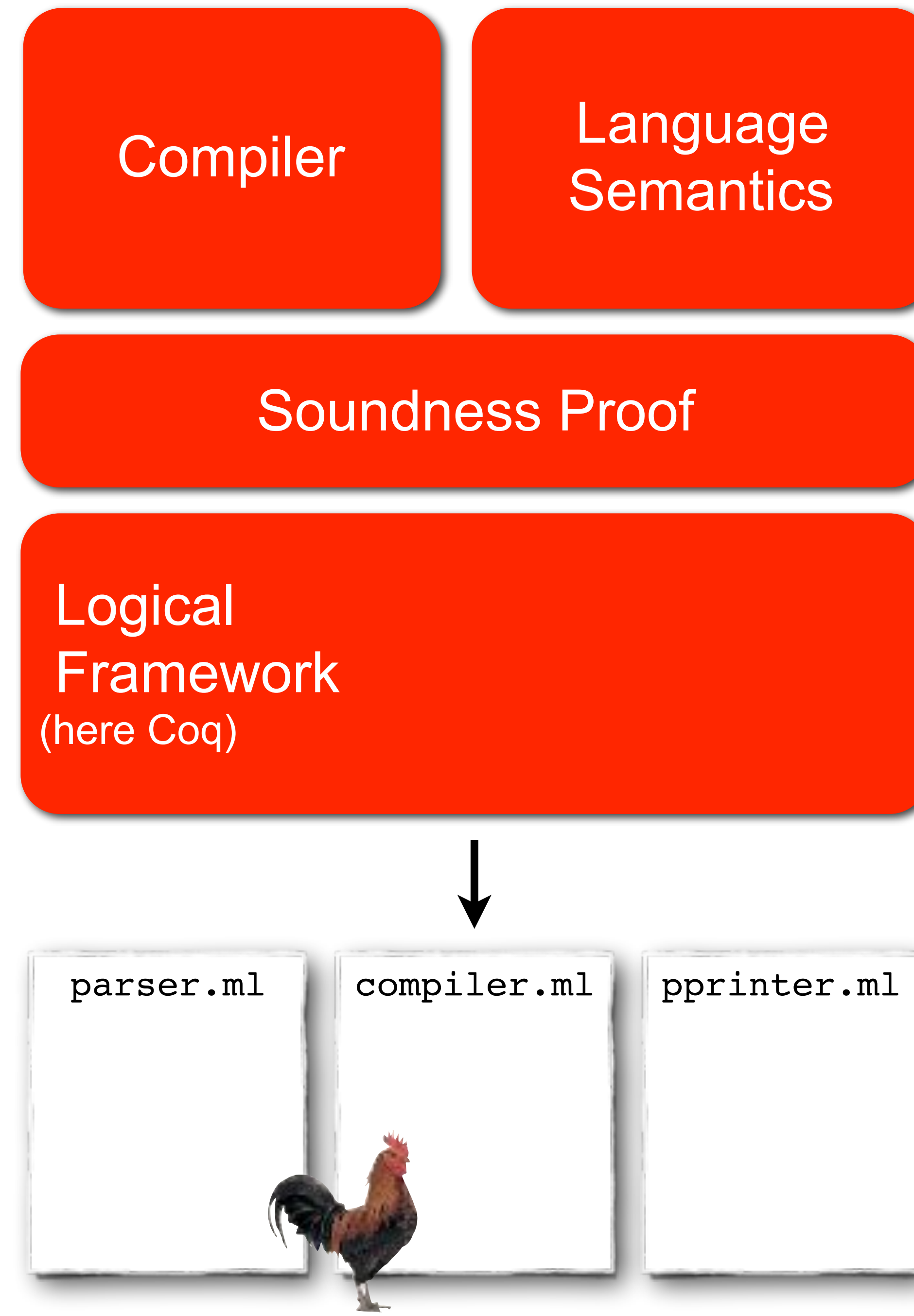
```
Theorem compiler_is_correct :
∀ S C, compiler S = OK (C) → safe (S) →
      «C behaves like S».
```

We interactively and mechanically prove this theorem

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

We extract an OCaml implementation of the compiler.

```
Extraction compiler.
```

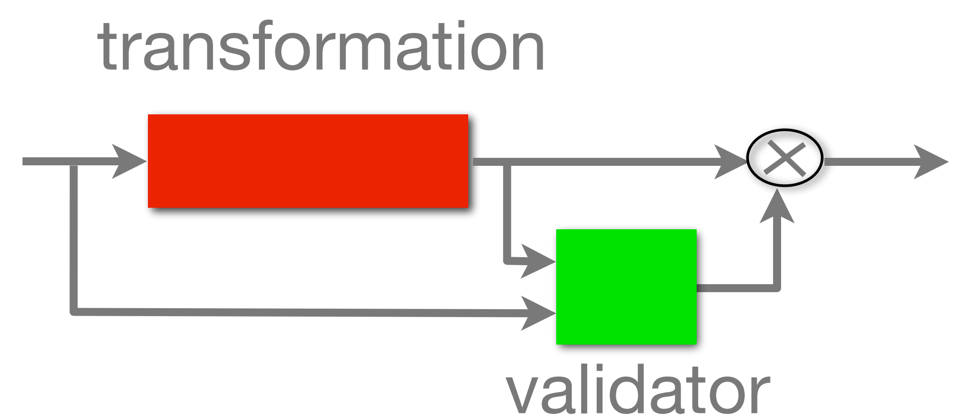


Verification patterns
(for each compilation pass)

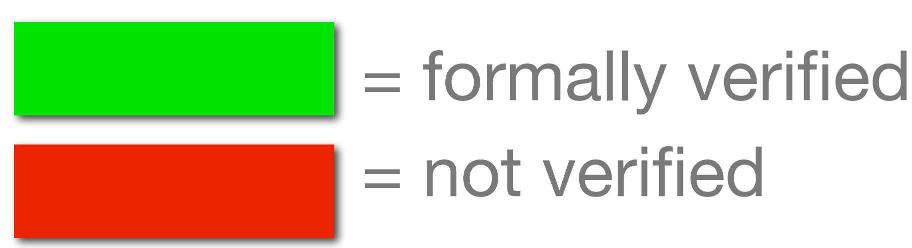
Verified transformation

Proved once for all

Verified translation validation



One proof per program to compile



Verasco

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.

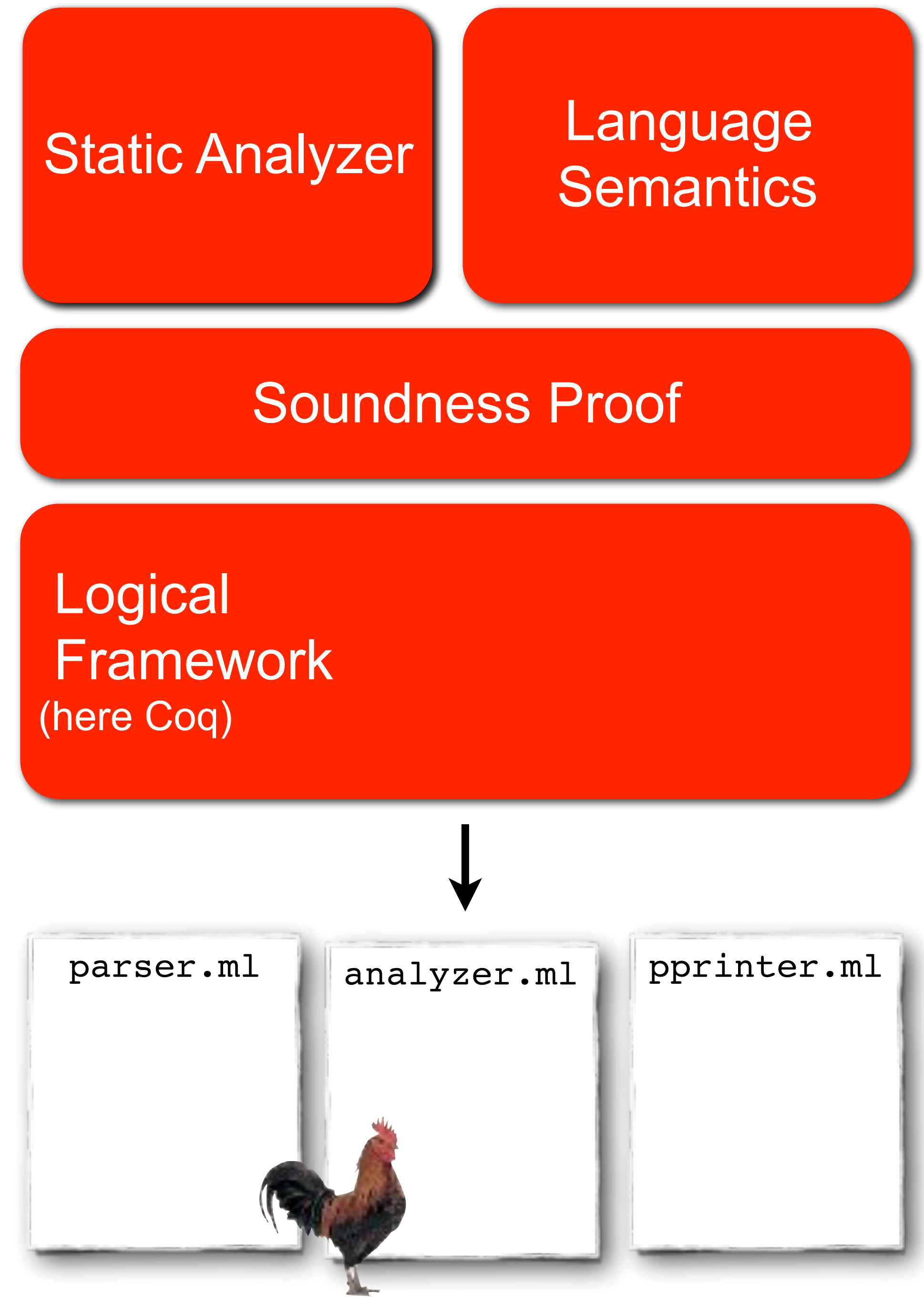
```
Theorem analyzer_is_sound :
 ∀ P, analyzer P = Yes →
      safe(P).
```

We interactively and mechanically prove this theorem

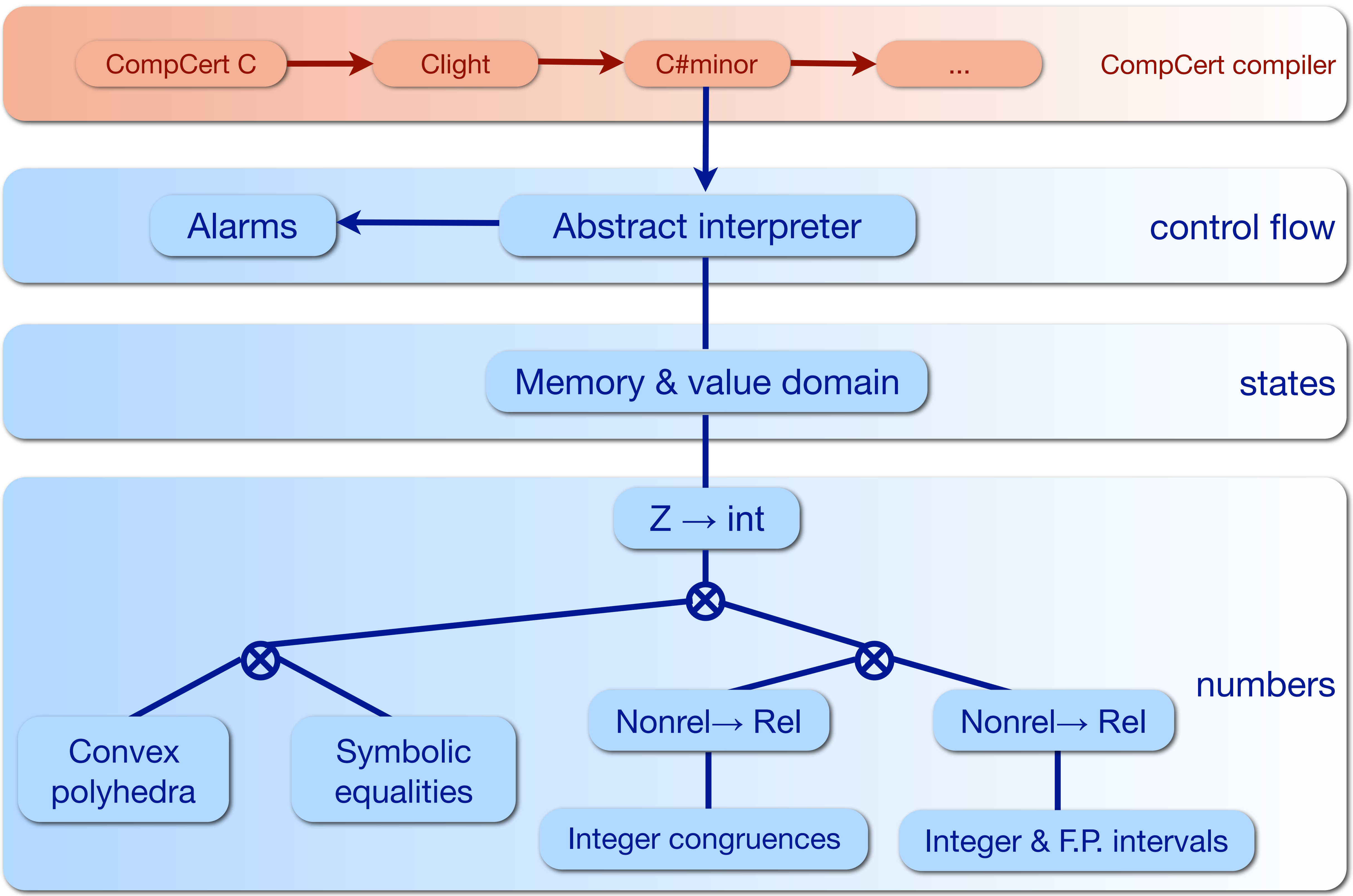
```
Proof. ... (* a few months later *) ...
Qed.
```

We extract an OCaml implementation of the analyzer.

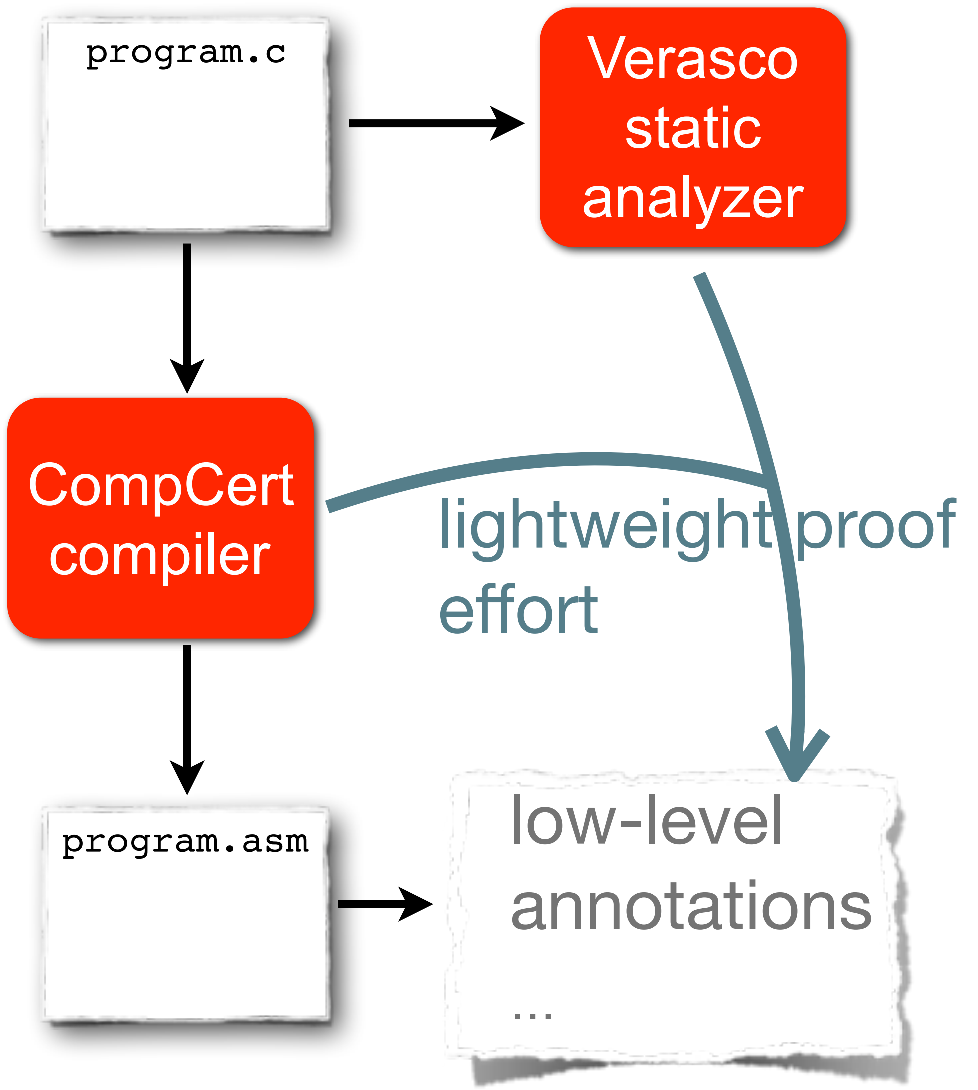
```
Extraction analyzer.
```



General architecture of Verasco

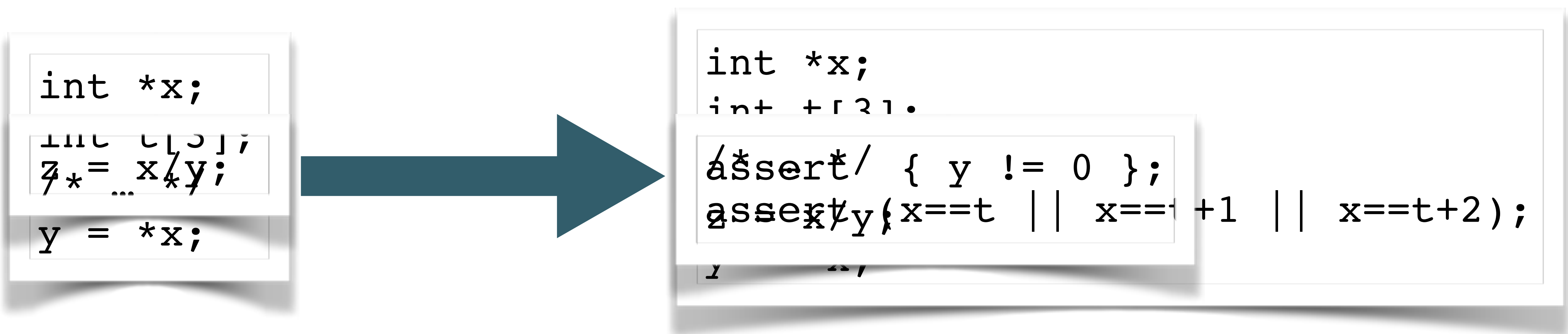


A lightweight methodology to correctly translate the results of static analysis



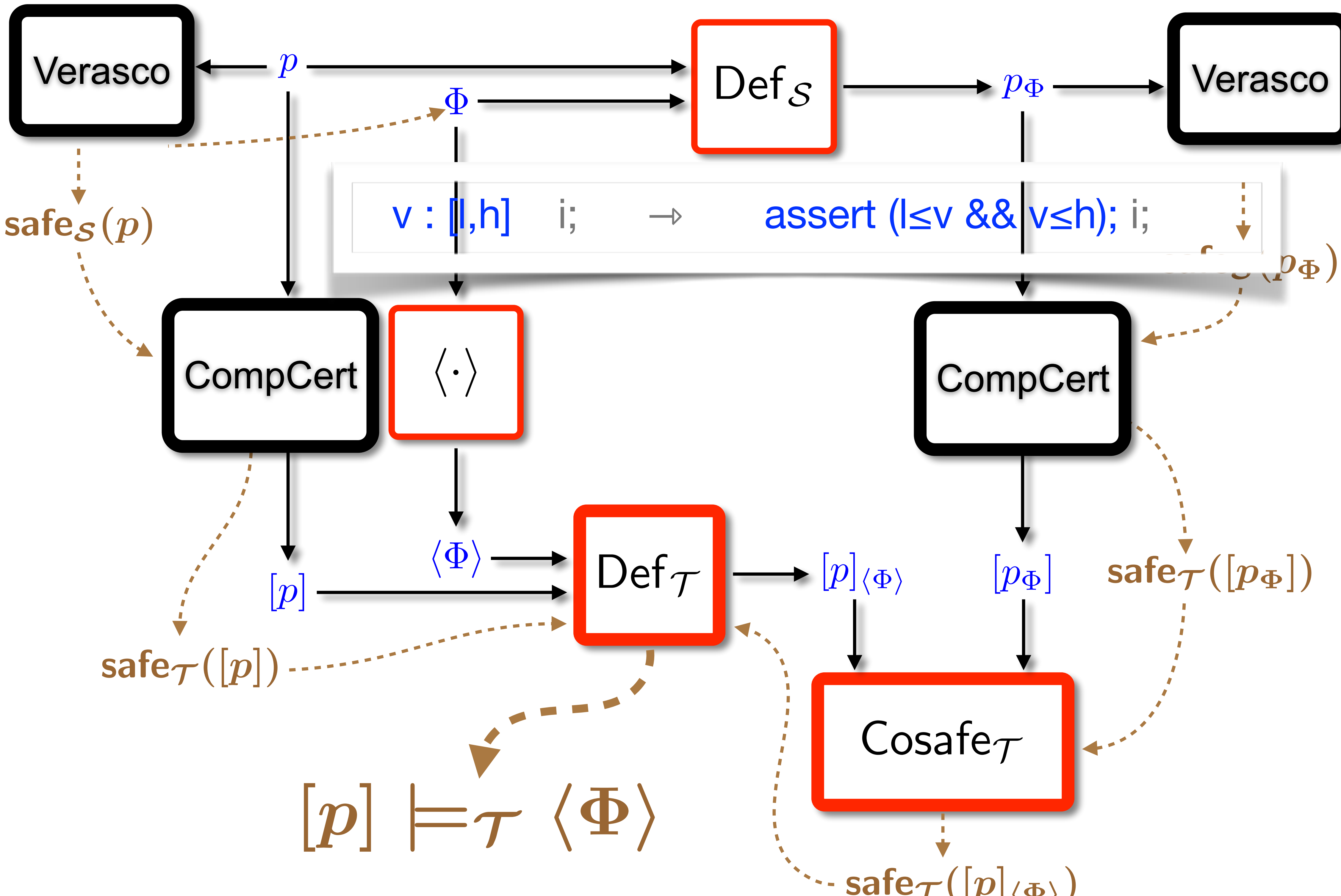
Key ideas

- 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.



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

Methodology



Use case:
cryptographic
constant-time

Secure cryptography ?

Lots of secure crypto algorithms were produced in the past decades:
AES, RSA, SHA1/2/3, ECDSA (used in Bitcoin).

Attacks against these algorithms were also published:

- Tromer, Osvik, Shamir, 2006: Attack against AES, recover the secret key from Linux's encrypted partitions in 65ms.
- Yarom, Falkner, 2014: Attack against RSA in GnuPG 1.4.13. « On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round. »
- Benger, van de Pol, Smart, Yarom, 2014: Attack on ECDSA using the secp256k1 curve « We demonstrate our analysis via experiments using the curve secp256k1 used in the Bitcoin protocol. In particular we show that with as little as 200 signatures we are able to achieve a reasonable level of success in recovering the secret key for a 256-bit curve. »

Timing attacks

- Secret data can have influence on running time of the program
- Attacker measures timing
- Attacker infers what is required to recover secret

Examples

```
if (secret)
  foo();
else
  bar();
```

Computes a^b mod n, where b is a secret key in RSA
pseudo-code from Wikipedia

```
function modular_pow(base, exponent, modulus)
    if modulus = 1 then return 0
    Assert :: (modulus - 1) * (modulus - 1) does not OVF base
    result := 1
    base := base mod modulus
    while exponent > 0
        if (exponent mod 2 == 1):
           result := (result * base) mod modulus
        exponent := exponent >> 1
        base := (base * base) mod modulus
    return result
```

leaks how many bits of exponent are 1

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

- Leakage model: conditional branchings and memory accesses can leak information
  Ex.: Leakage ($\langle \sigma$, if e then p1 else p2$\rangle$) = σ (e)

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

Conclusion

Approach to formally verify translation of static analysis results in a formally verified compiler

- lightweight proof-effort
- reduces security to safety
- improves a previous security analysis at pre-assembly level

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)

Questions ?

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.