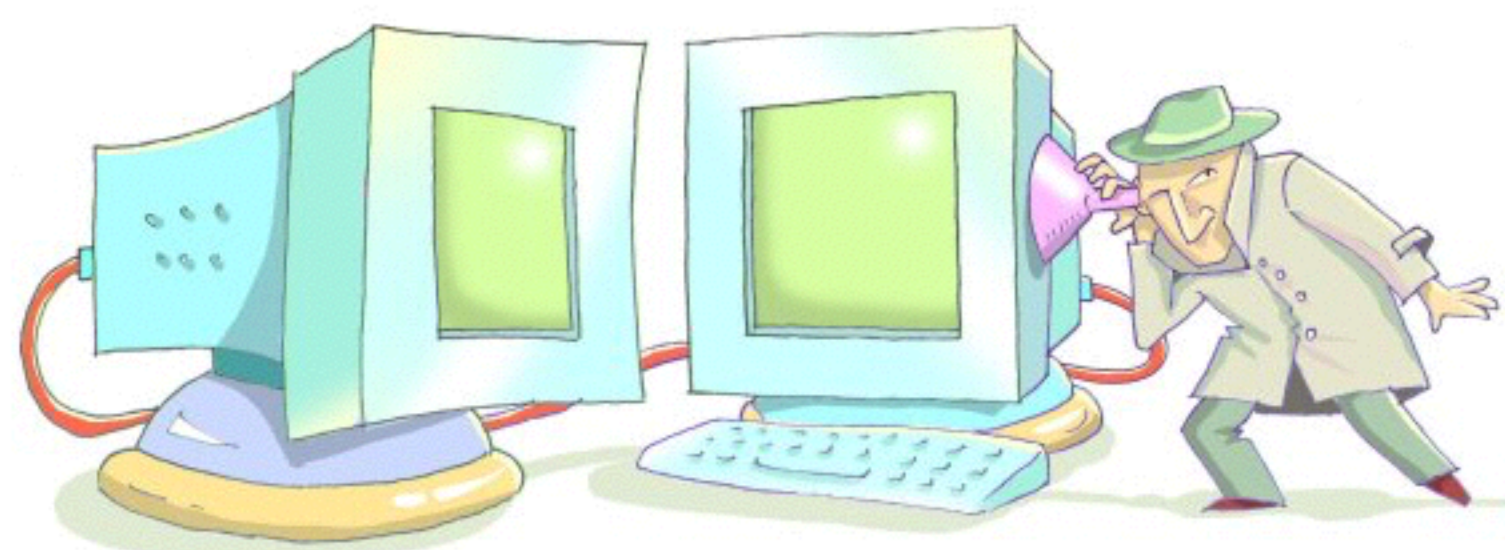
Verification of constant-time implementation in the Compcert compiler toolchain

David Pichardie

Cache timing attacks

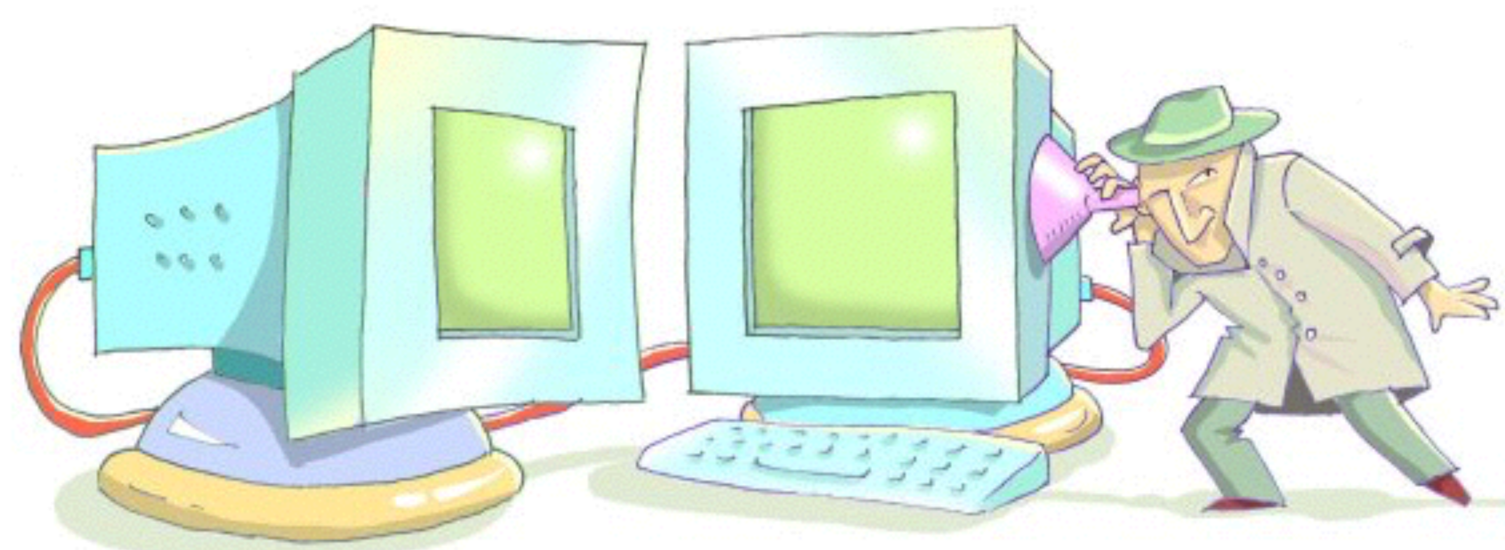
- Common side-channel: Cache timing attacks
- Exploit the latency between cache hits and misses
- Attackers can recover cryptographic keys
  - Tromer et al (2010), Gullasch et al (2011) show efficient attacks on AES implementations
- Based on the use of look-up tables
  - Access to memory addresses that depend on the key



Constant-time programs

Characterization

- Constant-time programs do not:
  - branch on secrets
  - perform memory accesses that depend on secrets
- There are constant-time implementations of many cryptographic algorithms: AES, DES, RSA, etc



Verification of constant-time programs

Challenges

- Provide a mechanism to formally check that a program is constant-time
    - static tainting analysis for implementations of cryptographic algorithms
- At low level implementation (C, assembly), advanced static analysis is required
    - secrets depends on data, data depends on control flow, control flow depends on data…
- A high level of reliability is required
    - semantic justifications, Coq mechanizations…
- Attackers exploit executable code, not source code
    - we need guaranties at the assembly level using a compiler toolchain

Background: verifying a compiler

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

= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves 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,
> then «C behaves like S».

Background: verifying a compiler

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

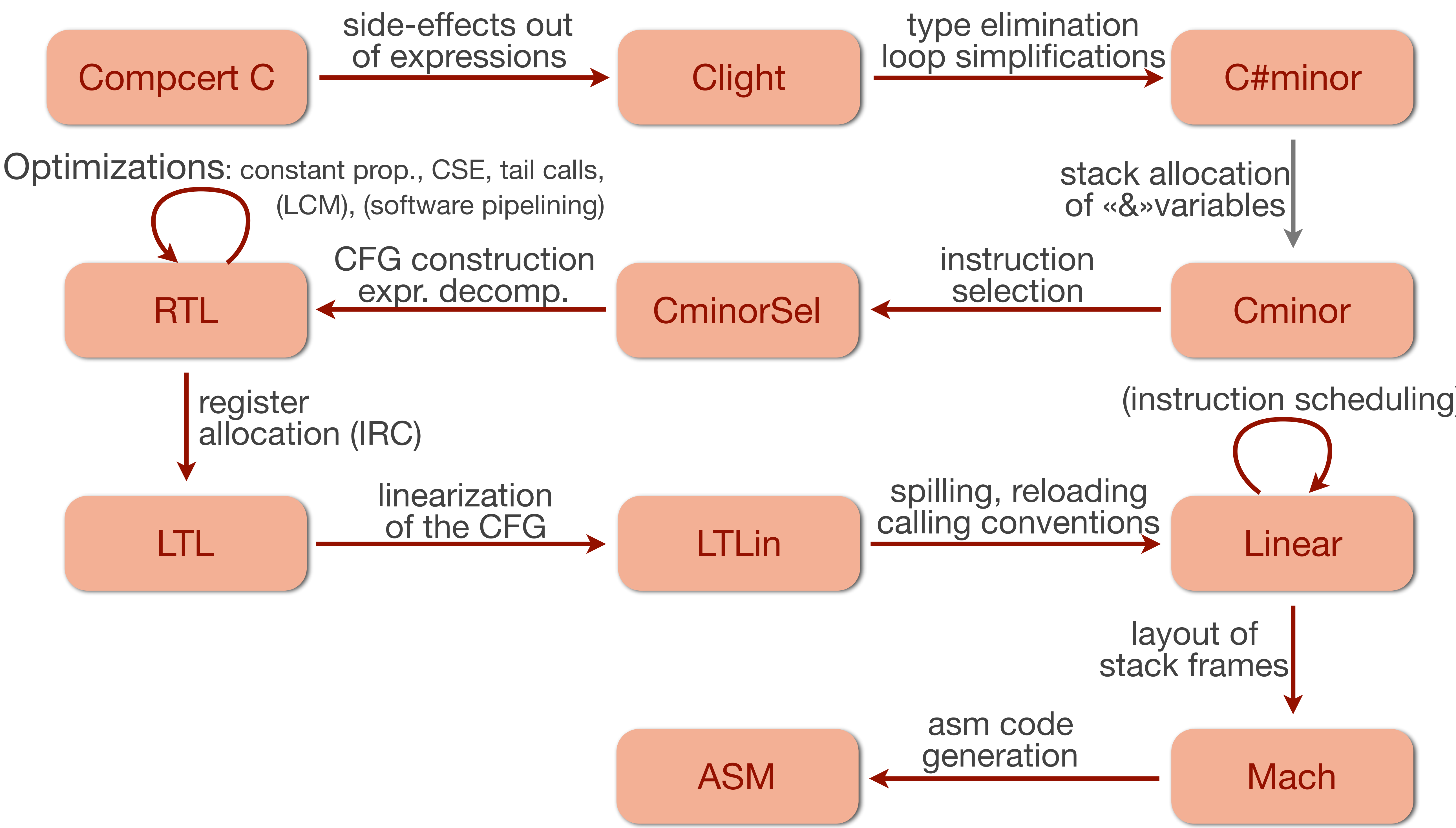
= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

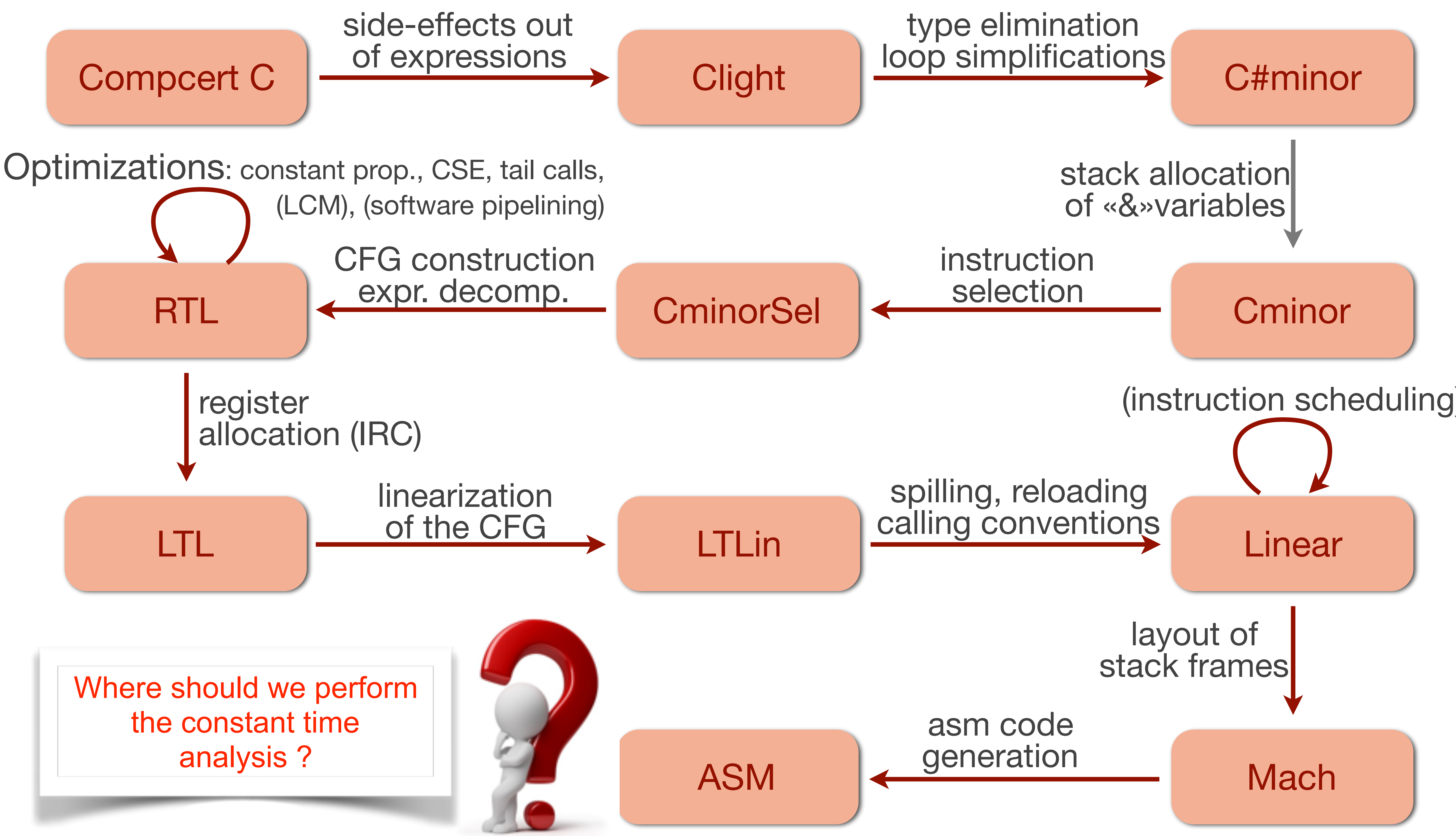
> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> then «C behaves like S».

does not deal with the
constant-time security property !

CompCert: 1 compiler, 11 languages



CompCert: 1 compiler, 11 languages



This talk: 3 approaches

1. Analysis at (almost) assembly level

G. Barthe, G. Betarte, J. D. Campo, C. Luna and D. Pichardie.
System-level non-interference for constant-time cryptography.
CCS 2014.

2. Analysis at (almost) assembly level, with help from an analysis at source level

G. Barthe, S. Blazy, V. Laporte, D. Pichardie, A. Trieu.
Lightweight, Verified Translation Validation of Static Analyses.
CSF 2017.

3. Analysis at source level

Sandrine Blazy, David Pichardie, Alix Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017.

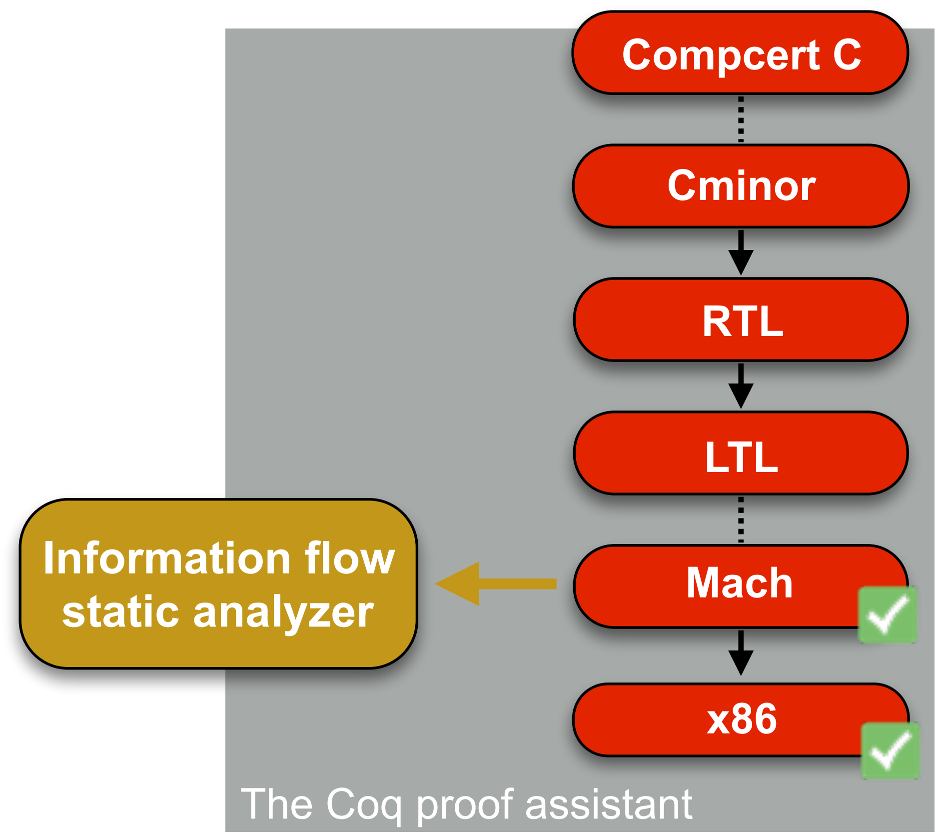
First approach
Performing the analysis at (pre)-assembly level

Good place for proving constant-time on actual implementation

- Compcert Mach level is the last IR before full assembly
- Compcert does no introduce new memory operations after that level

But the place is challenging for static analysis tool

- no more memory abstraction: memory is one single big array
- all memory accesses handle some kind of arithmetic on adresses



First approach
Performing the analysis at (pre)-assembly level

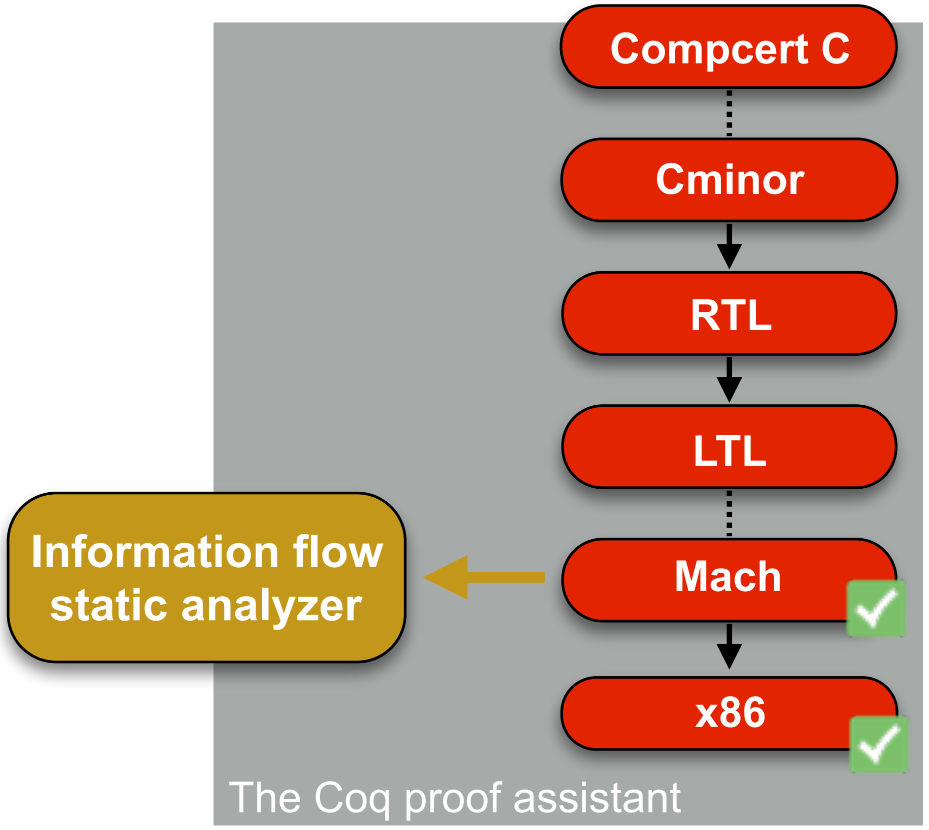
G. Barthe, G. Betarte, J. D. Campo, C. Luna and D. Pichardie.
System-level non-interference for constant-time cryptography.
CCS 2014.

Strong points

- verified static alias analysis at Mach level
- verified taint analysis using the alias information
- several experiments on real crypto C programs: Salsa20, Sha256, TEA

Weak points

- several manual rewriting of the source programs are required
- efficiency is bad because of function full inlining



Performing the analysis at (pre)-assembly level

Technical details

Low level memory model

- registers + one memory block for each global variable + one memory block for the whole stack

Pre-analysis

- we perform a points-to analysis to tracks the set of memory blocks manipulated by each memory instruction

Taint analysis

- one taint for each global variable
- one taint for each register, at each program point
- one taint for each stack slot (byte), at each program point

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

$$\frac{\begin{array}{c} p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \\ \tau(\vec{r}) = \mathsf{Low} \qquad \tau(r) \sqsubseteq X_h(\mathcal{S}) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{\begin{array}{c} p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

instruction at program point p

symbolic memory address

stored value

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \quad \tau(\vec{r}) = \mathsf{Low} \quad \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_h \vdash n : \tau \Rightarrow \tau}$$

size of the accessed memory block

next program point

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

instruction at program point p

symbolic memory address

stored value

points-to pre-analysis

size of the accessed memory block

next program point

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \quad \tau(\vec{r}) = \mathsf{Low} \quad \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

instruction at program point p

symbolic memory address

stored value

points-to pre-analysis

size of the accessed memory block

next program point

$$\frac{\begin{array}{c} p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \\ \tau(\vec{r}) = \mathsf{Low} \qquad \tau(r) \sqsubseteq X_h(\mathcal{S}) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{\begin{array}{c} p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

we forbid high taints on address computation

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

instruction at program point p

symbolic memory address

stored value

points-to pre-analysis

size of the accessed memory block

next program point

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \quad \tau(\vec{r}) = \mathsf{Low} \quad \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathsf{store}_{\varsigma}(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

we forbid high taints on address computation

if register r is high, global variable S must be high

Performing the analysis at (pre)-assembly level

Constraint based specification (excerpt)

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

taint of each global variable

program point

local types (registers + stack slot)

instruction at program point p

symbolic memory address

stored value

points-to pre-analysis

size of the accessed memory block

next program point

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \quad \tau(\vec{r}) = \mathsf{Low} \quad \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \quad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

we forbid high taints on address computation

if register r is high, global variable S must be high

we taint each stack position

Performing the analysis at (pre)-assembly level
Limitations

Engineering simplification

- no function call (we require full inlining)
- no dynamic allocation

Analysis precision limitation

- no array in stack (we only track constant adresses in stack)
- no fine grained struct tainting for structures in global variables

Manual rewriting

- every local arrays must be put as global!

But the analyser

- is proved correct and extracted from Coq formalisation
- runs on three real C programs

Program	LoC	Analysis time
TEA	70	0.08s
SHA256	419	68.14s
Salsa20	1077	0.68s

Second approach
Some help from higher level representations…

G. Barthe, S. Blazy, V. Laporte, D. Pichardie, A. Trieu.
Lightweight, Verified Translation Validation of Static Analyses.
CSF 2017.

Improvements

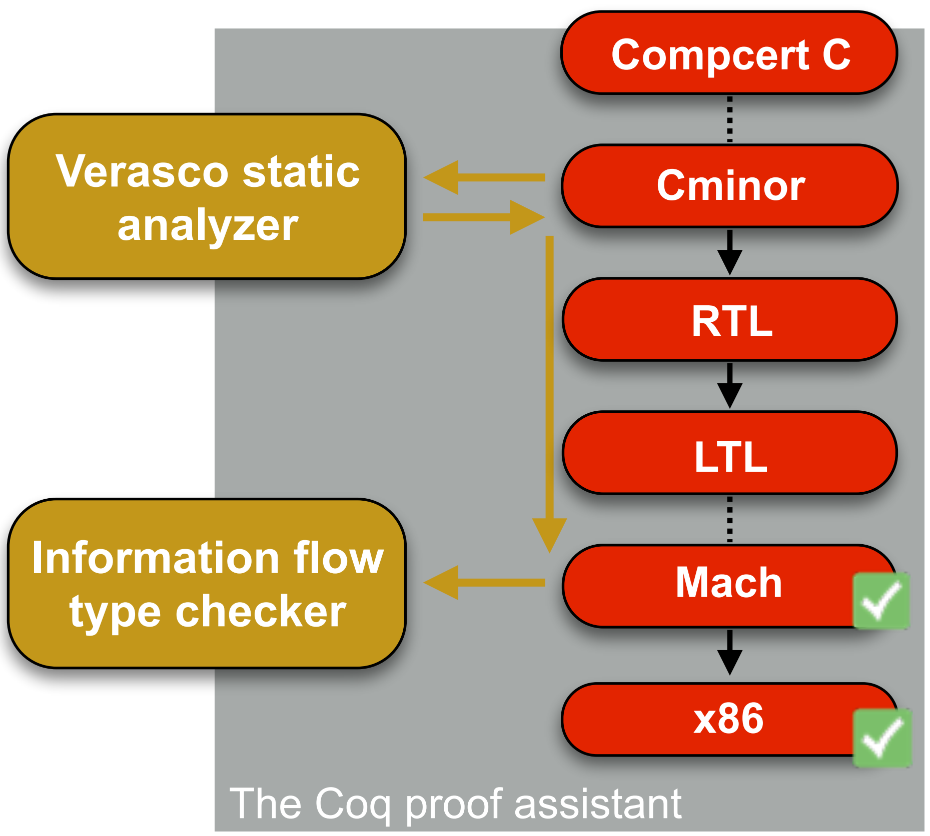
- no more manual rewriting
- better performance

How?

- The Verasco static analyser transmits strong alias informations trough the compiler toolchain

Extensibility

- soudness of the translation is independent of compiler optimisation passes



The Verasco project

INRIA Celtique, Gallium, Antique, Toccata + VERIMAG + Airbus
ANR 2012-2016

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation

- Language analyzed: the CompCert subset of C
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée's
- To prove the absence of undefined behaviors in C source programs

Slogan:

- if « CompCert ≈ 1/10th of GCC but formally verified »,
- likewise « Verasco ≈1/10th of Astrée but formally verified »

VERASCO

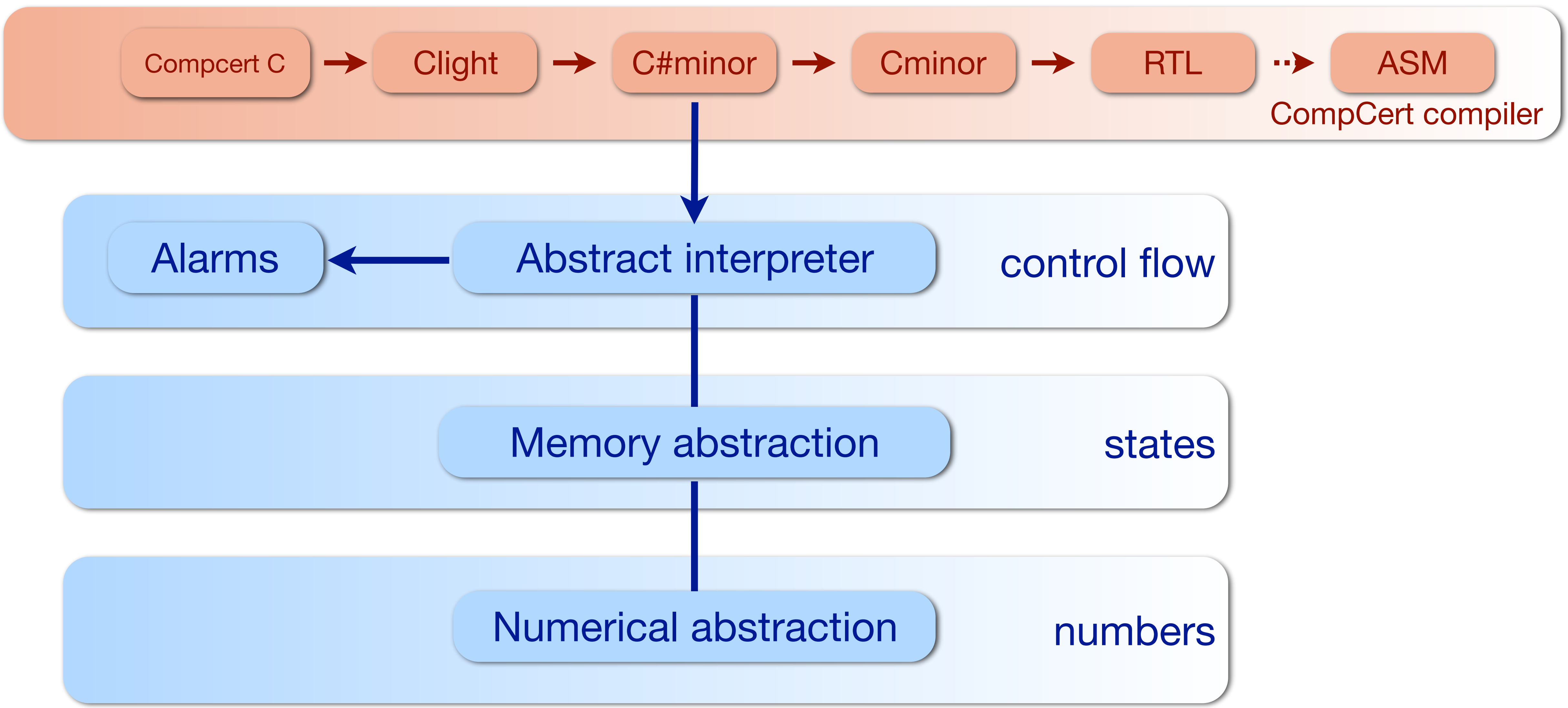
http://verasco.imag.fr

Verasco

A Formally-Verified C Static Analyzer

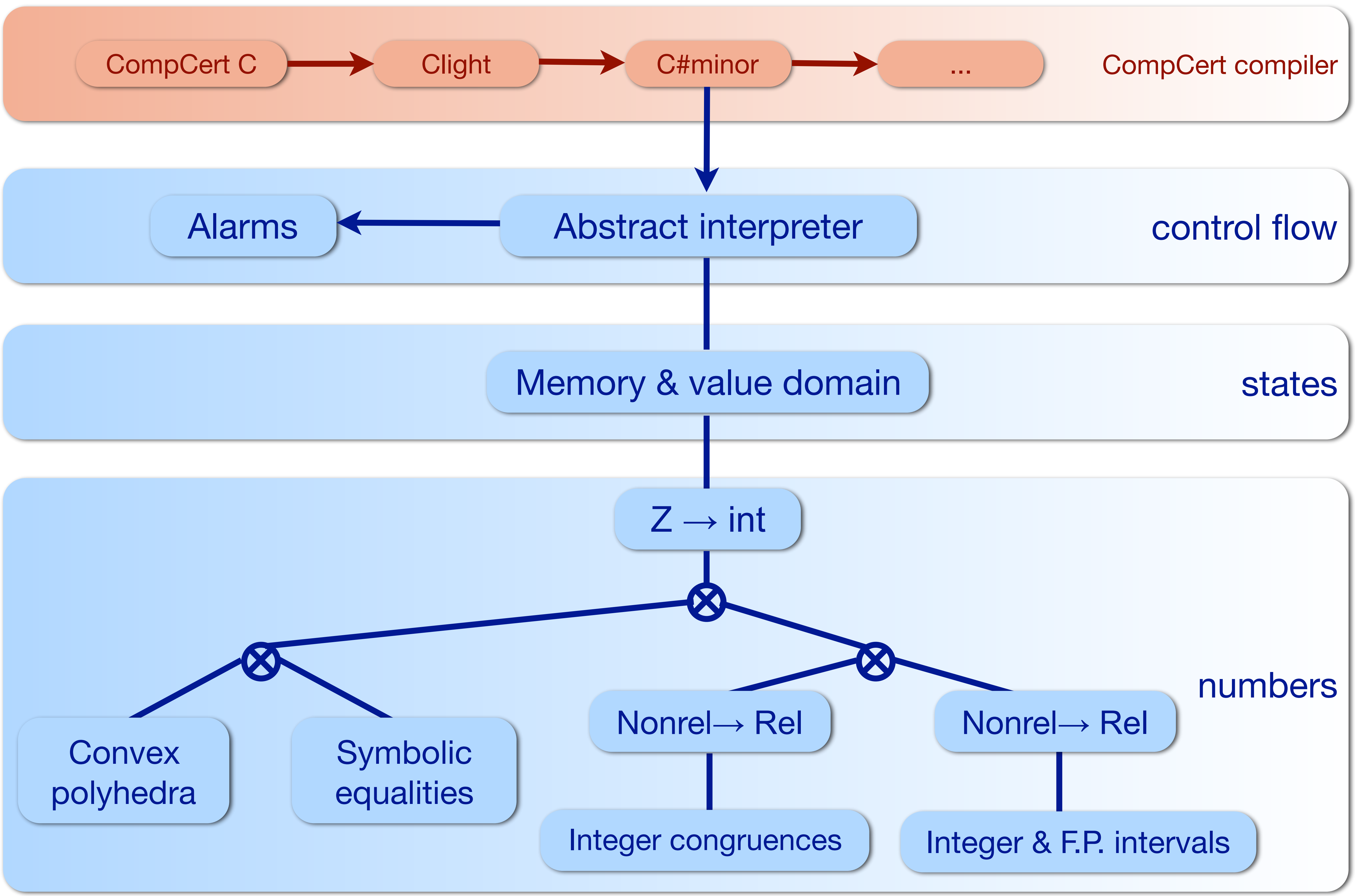
JH. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie.
A Formally-Verified C Static Analyzer.
POPL 2015.

S. Blazy, V. Laporte, D. Pichardie.
An Abstract Memory Functor for Verified C Static Analyzers.
ICFP 2016.



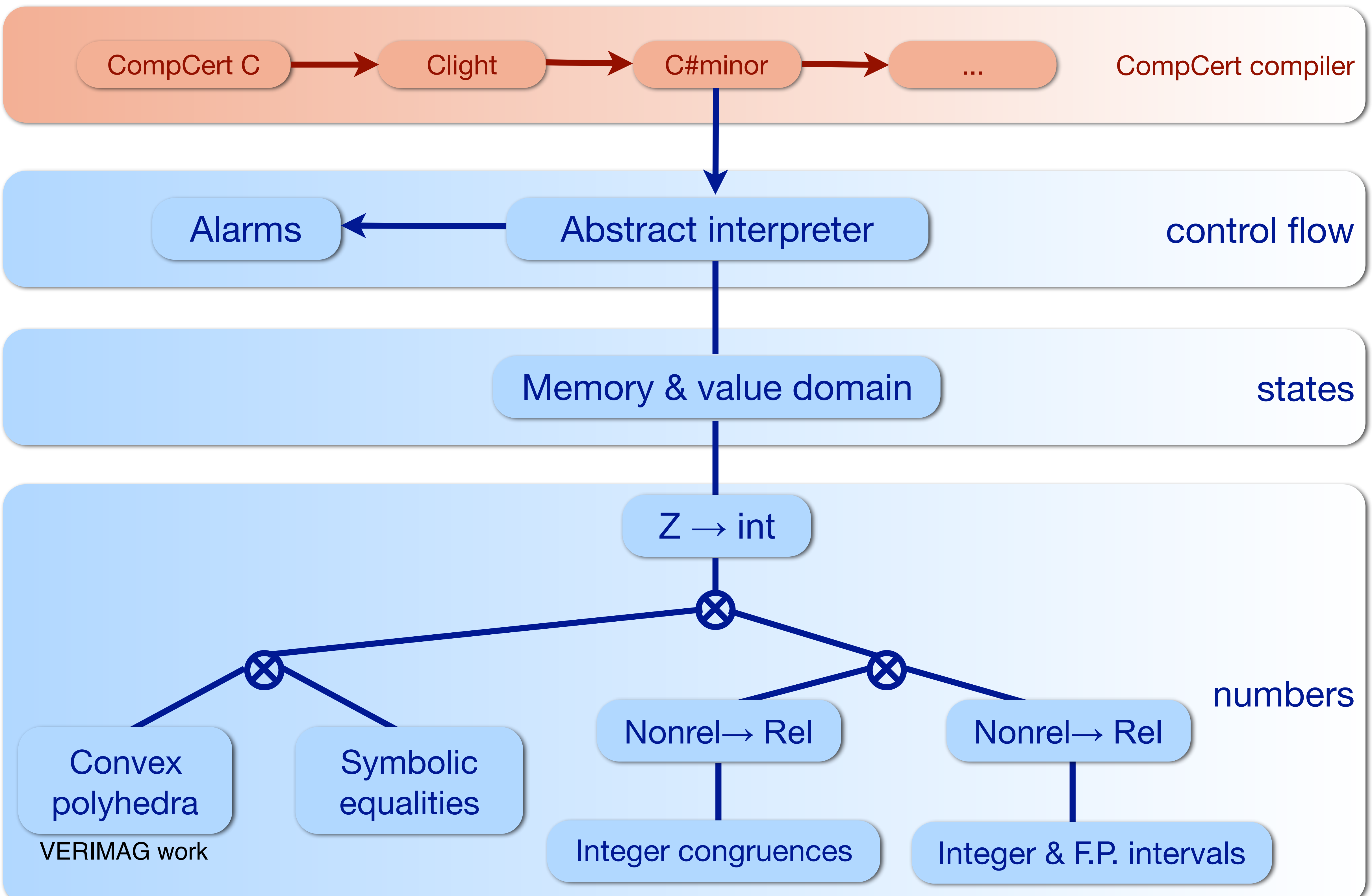
Verasco

Abstract numerical domains



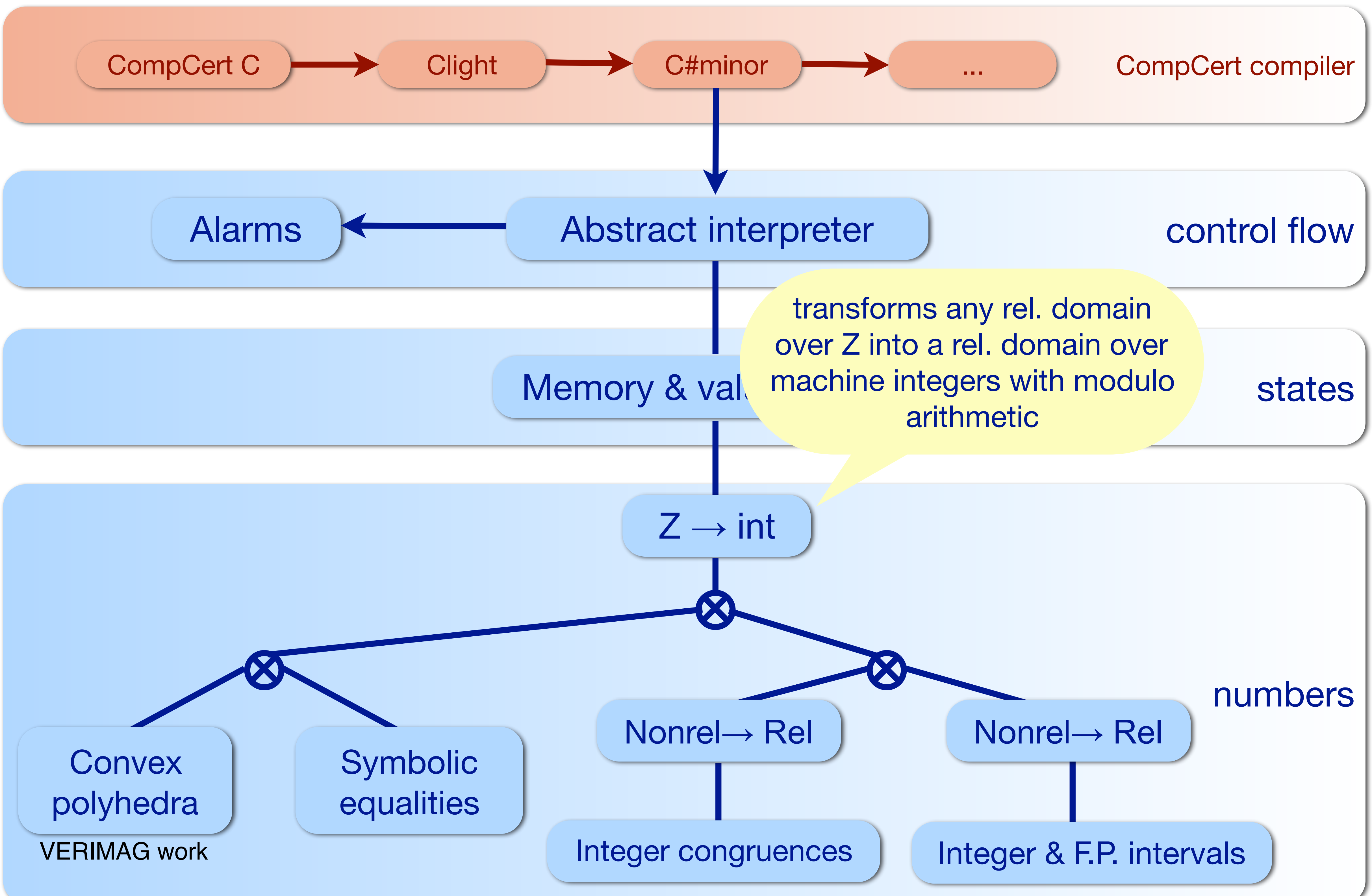
Verasco

Abstract numerical domains



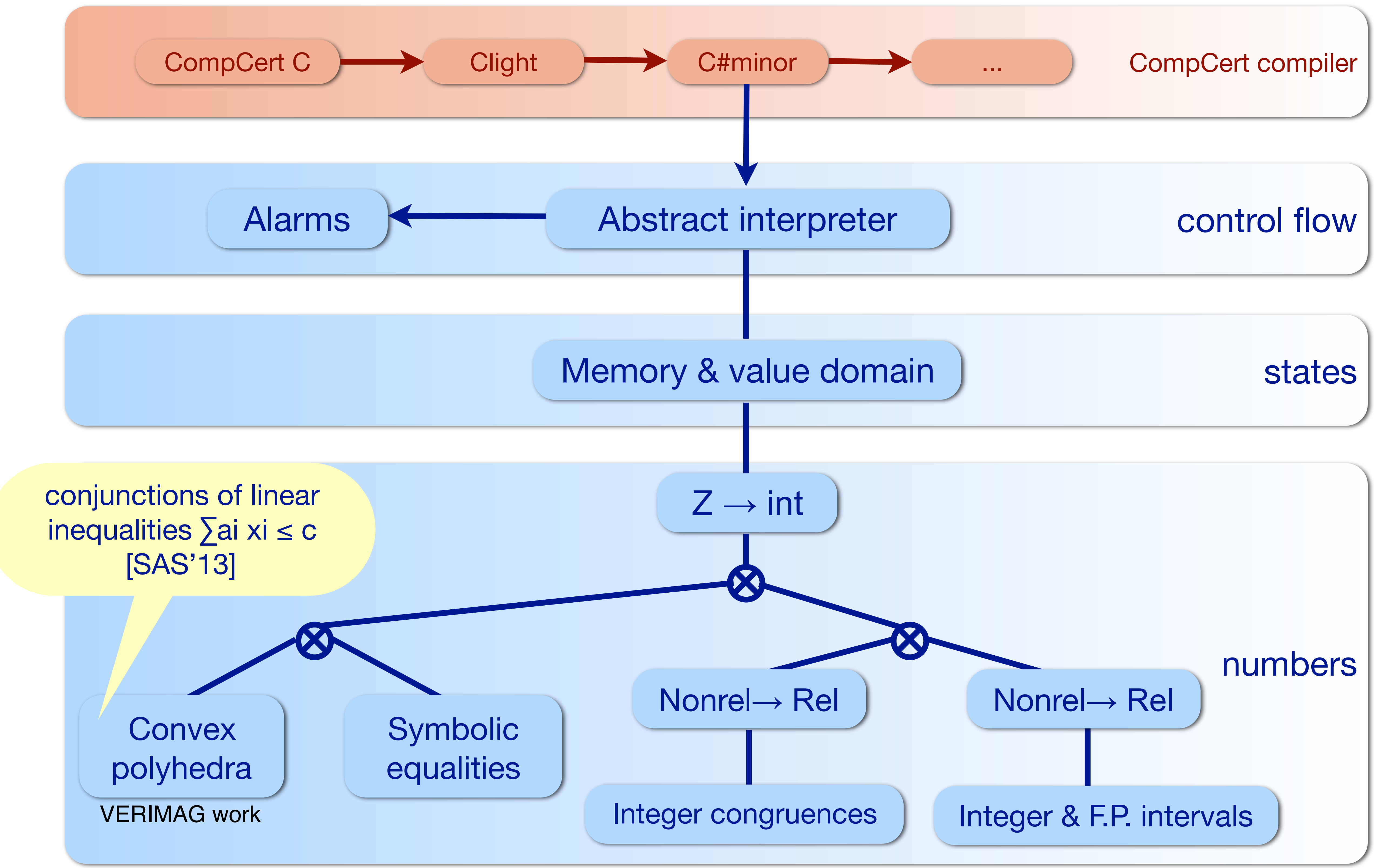
Verasco

Abstract numerical domains



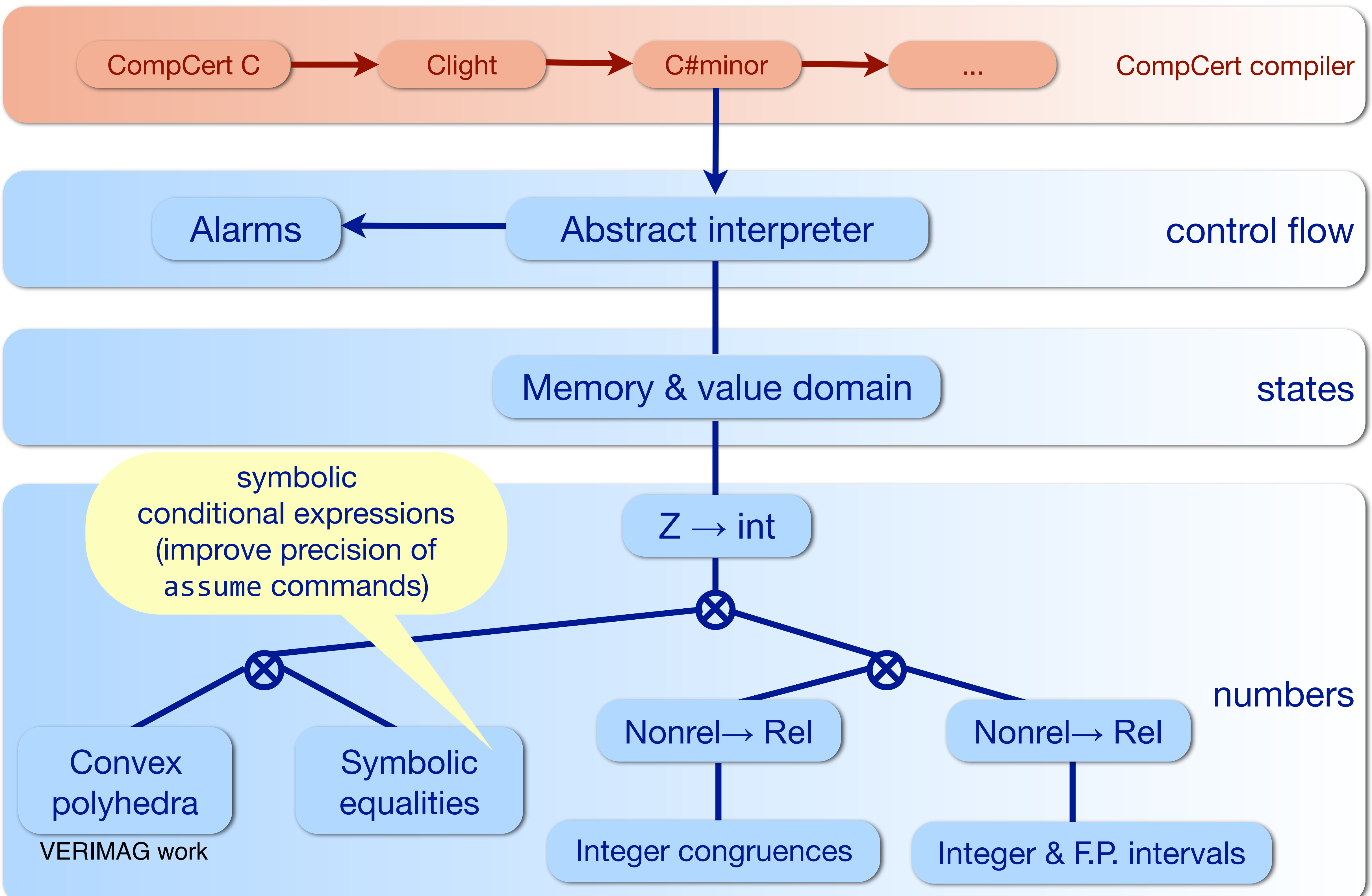
Verasco

Abstract numerical domains



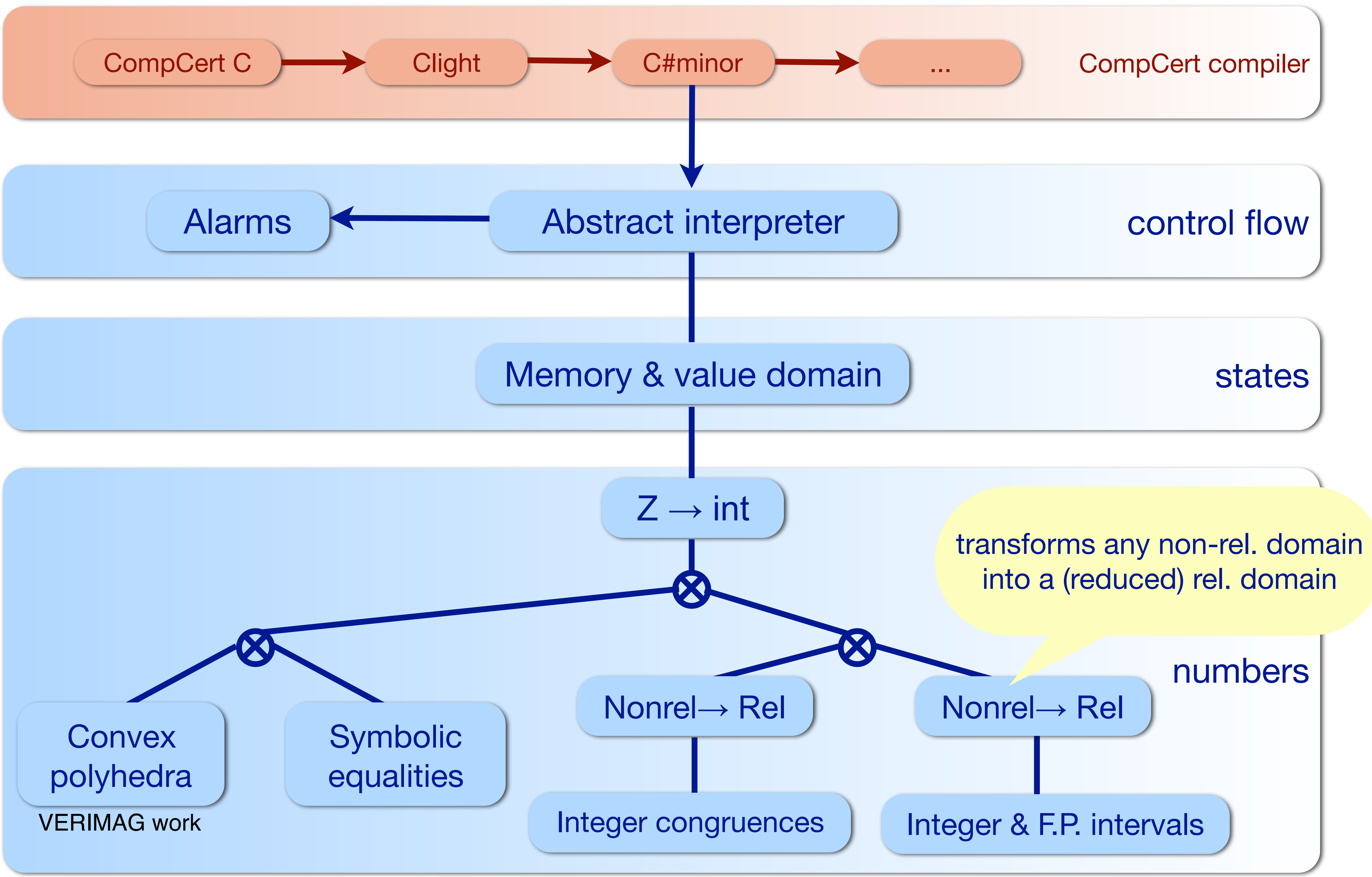
Verasco

Abstract numerical domains



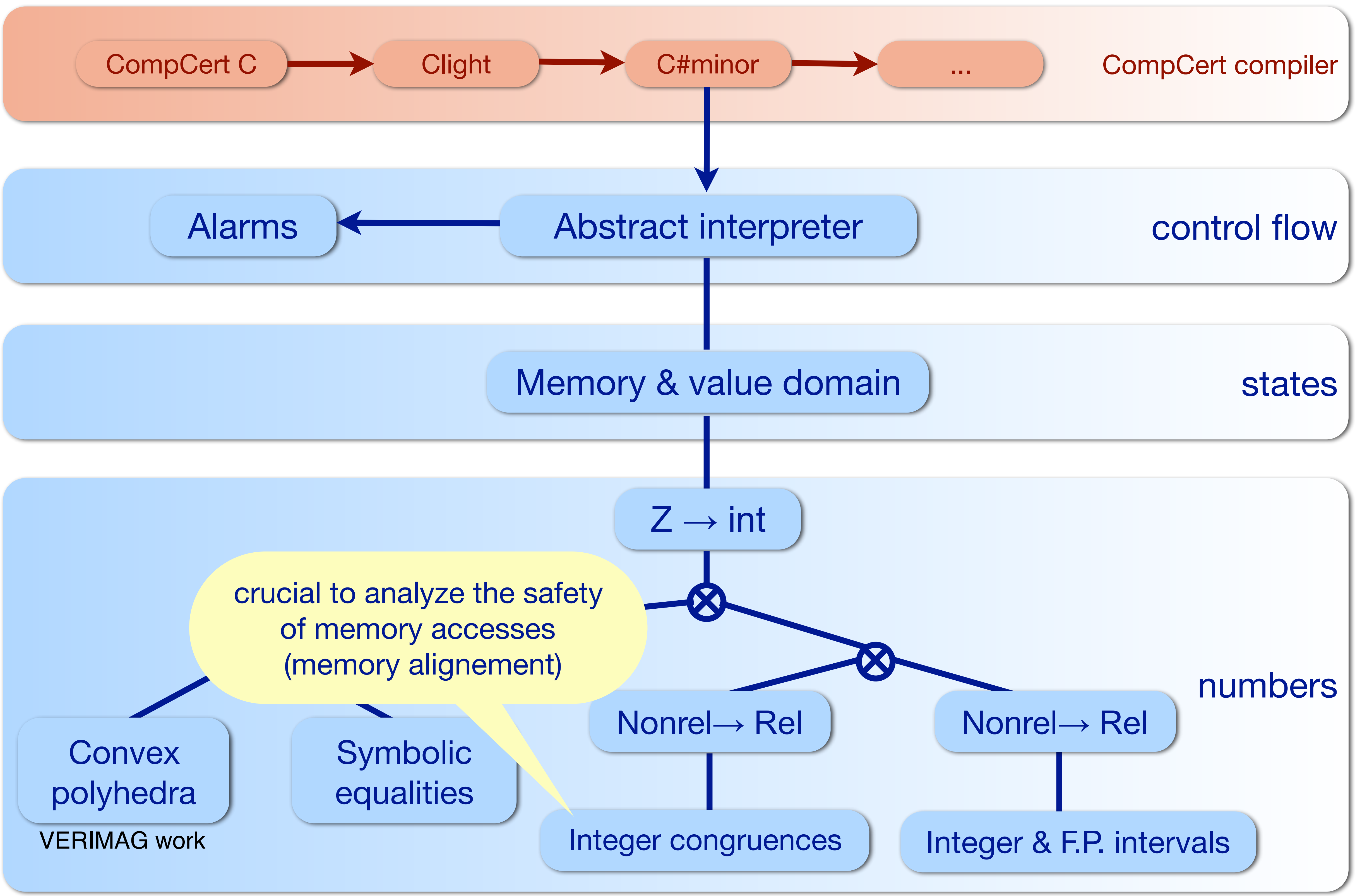
Verasco

Abstract numerical domains



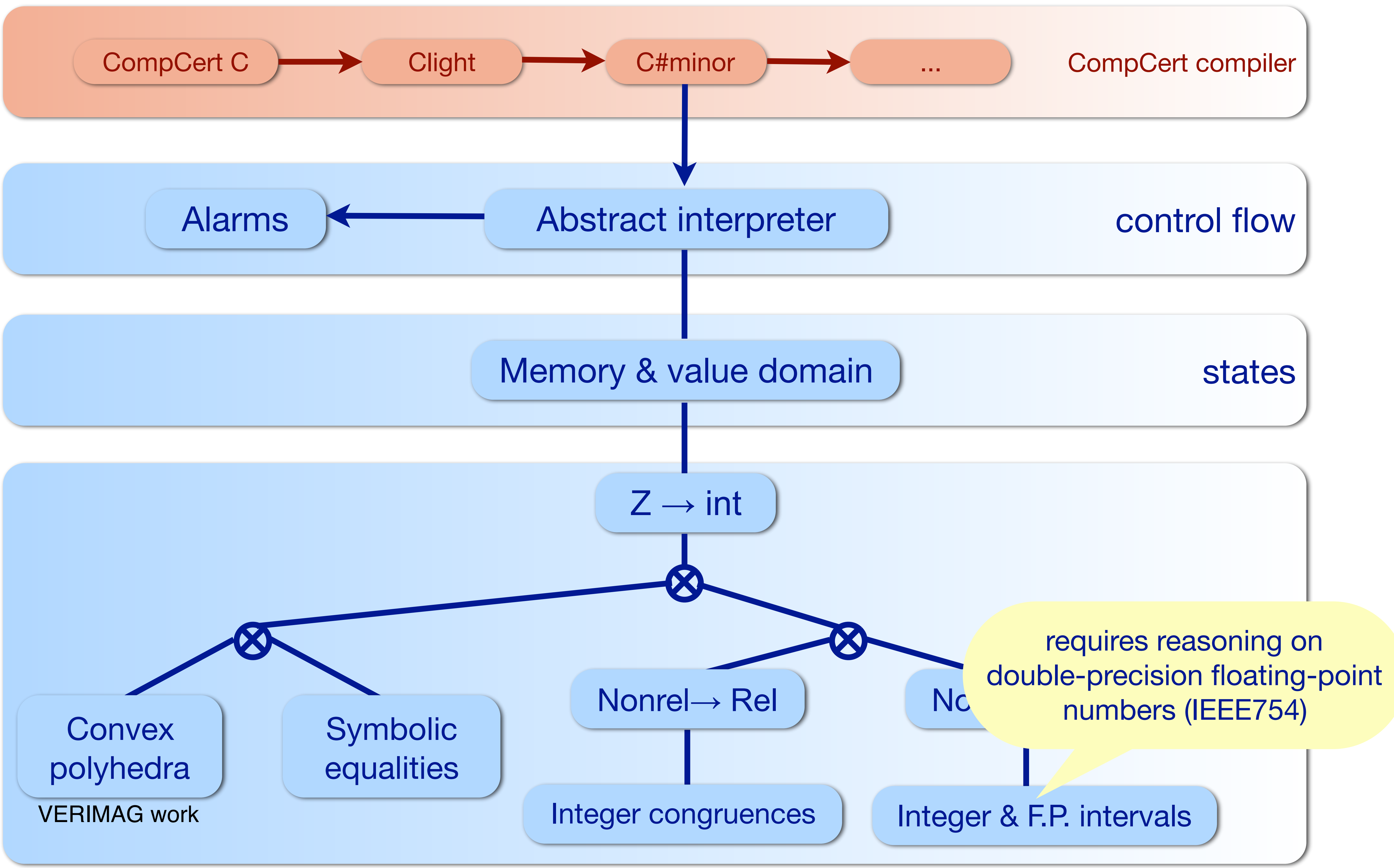
Verasco

Abstract numerical domains



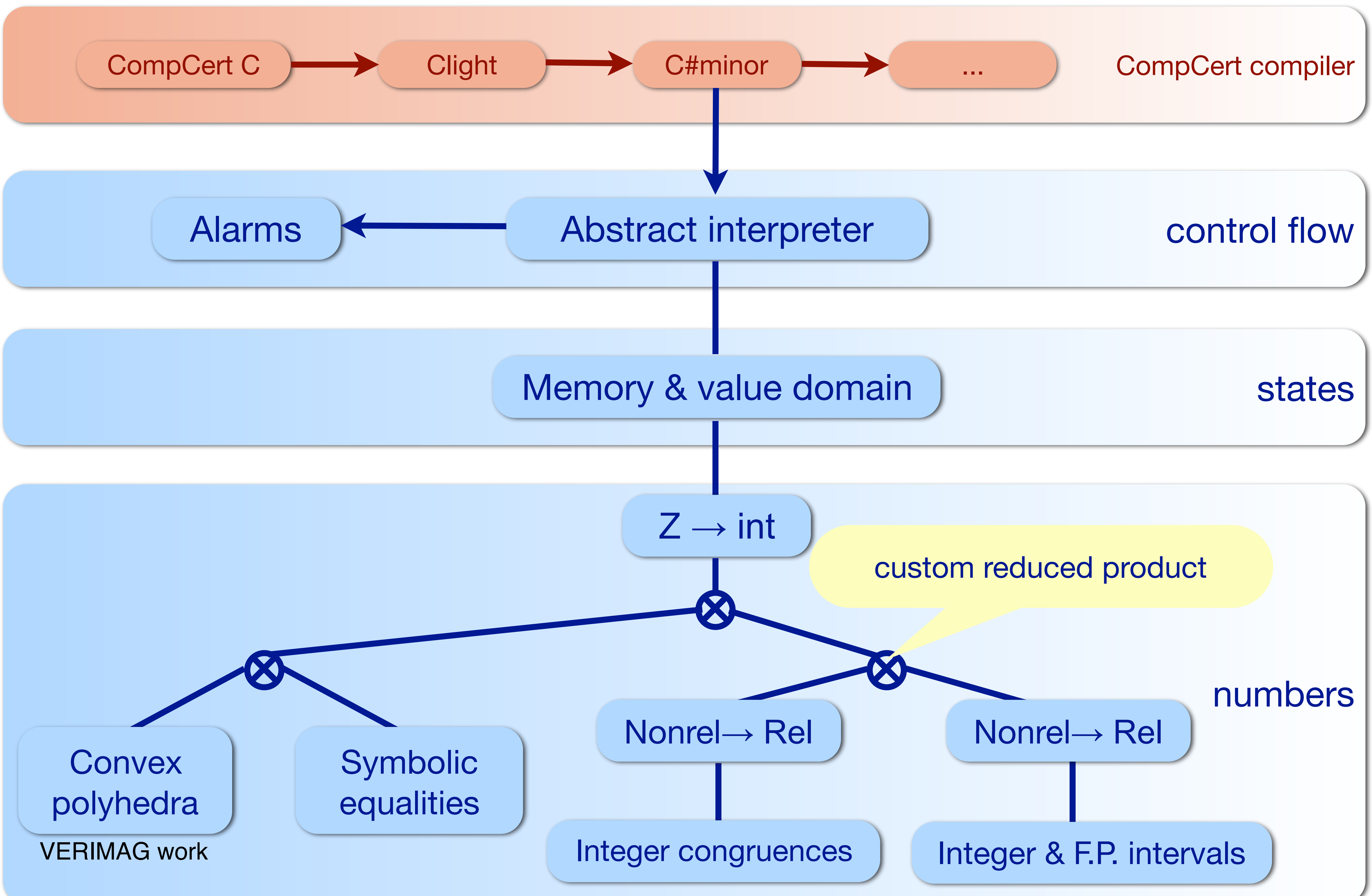
Verasco

Abstract numerical domains



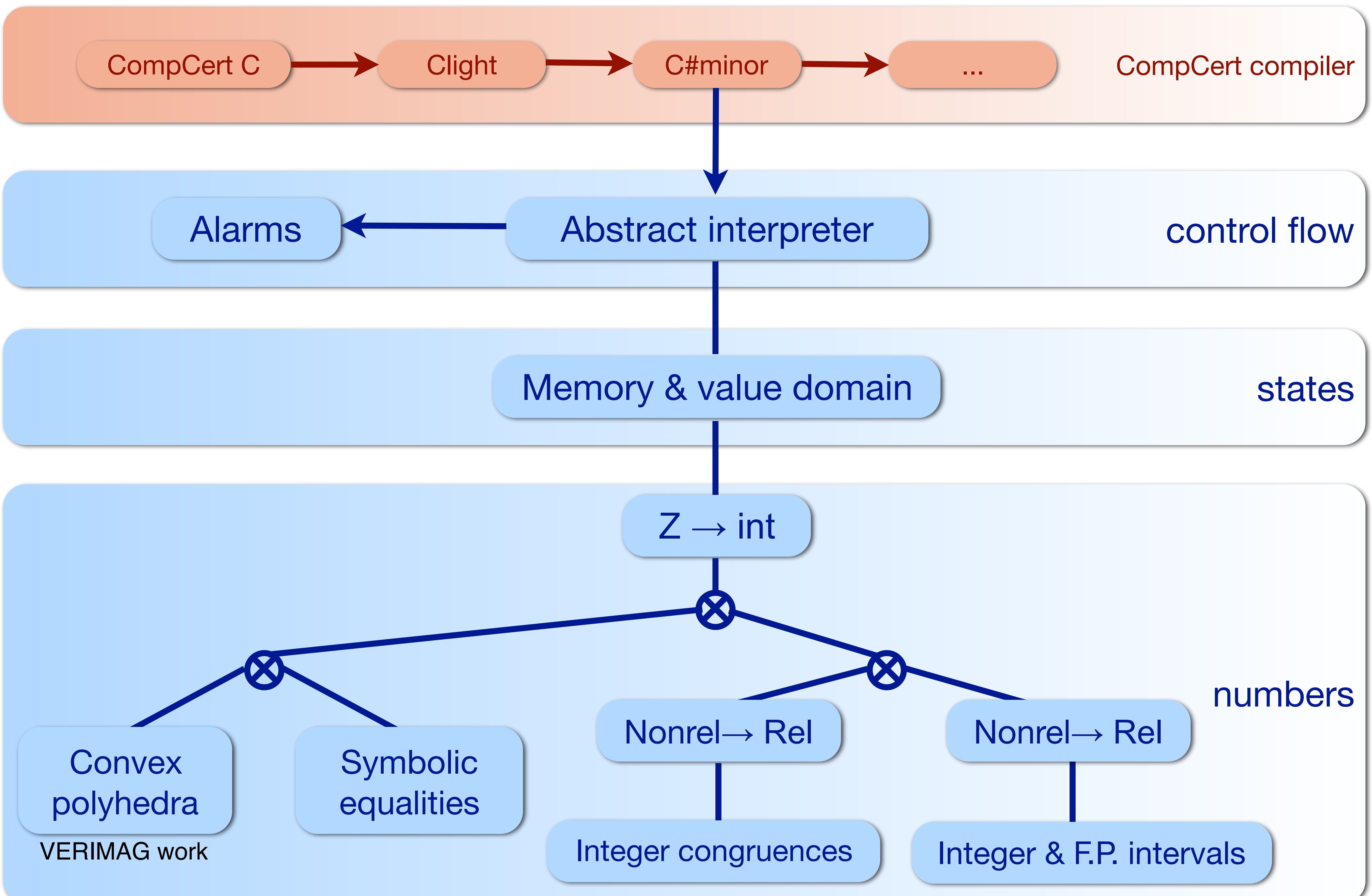
Verasco

Abstract numerical domains



Verasco

Abstract numerical domains



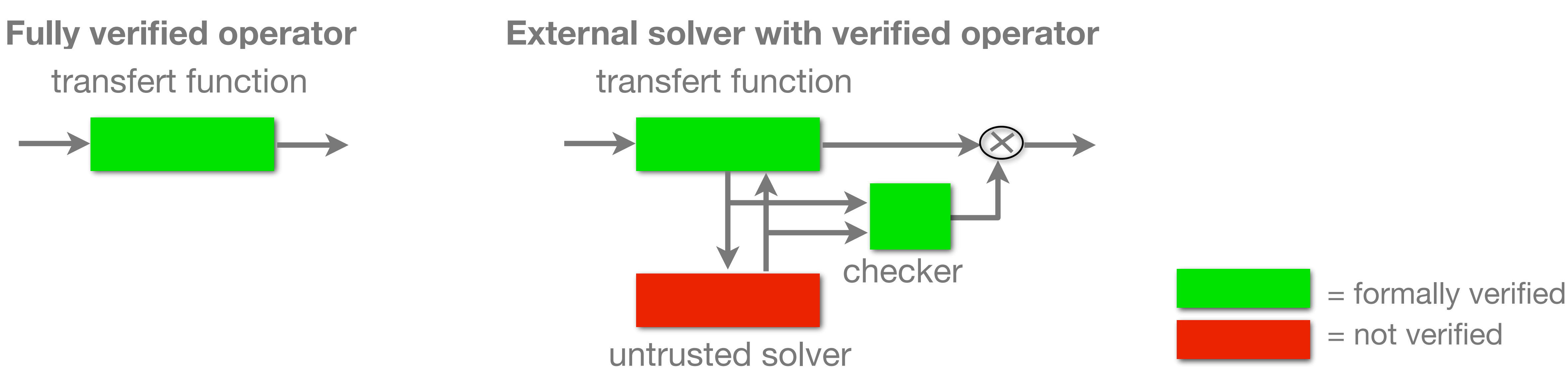
Verasco
Implementation

34 000 lines of Coq, excluding blanks and comments

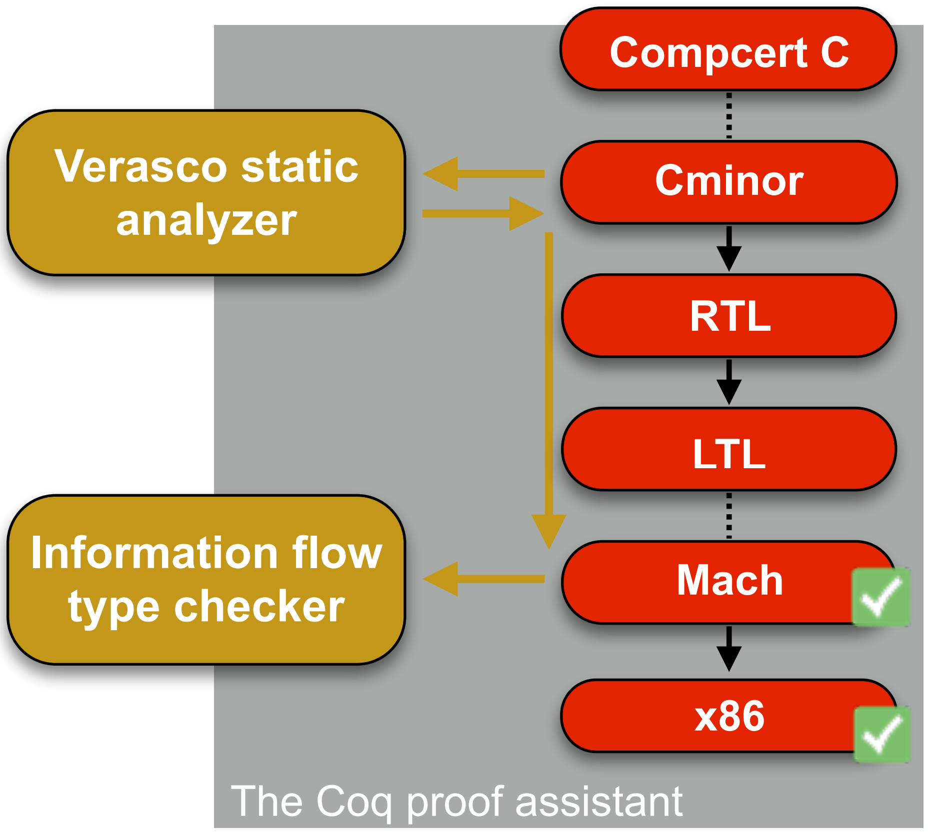
- half proof, half code & specs
- plus parts reused from CompCert

Bulk of the development: abstract domains for states and for numbers (involve large case analyses and difficult proofs over integer and floating points arithmetic)

Except for the operations over polyhedra, the algorithms are implemented directly in Coq's specification language.

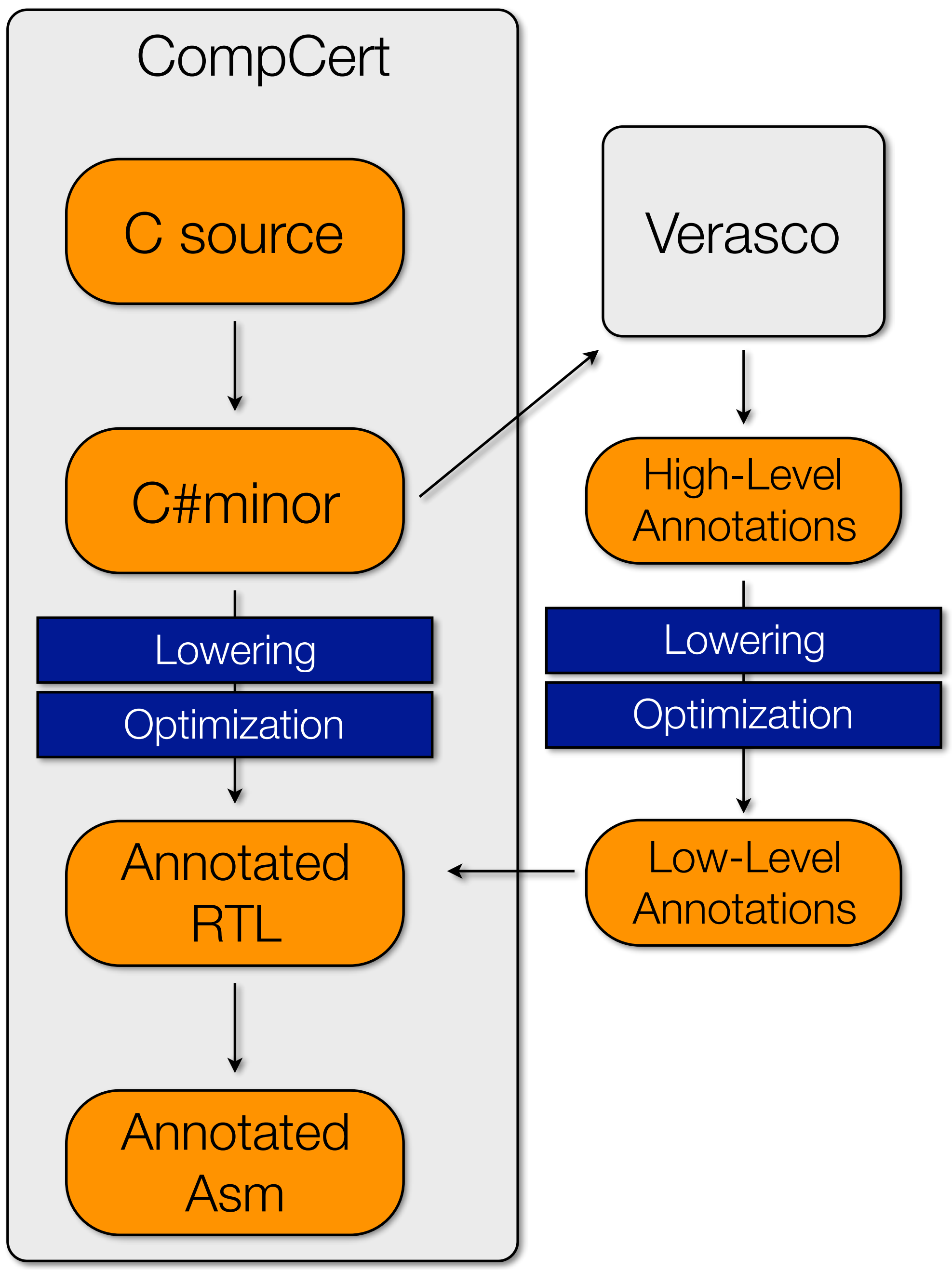


How to translate Verasco results downto assembly?

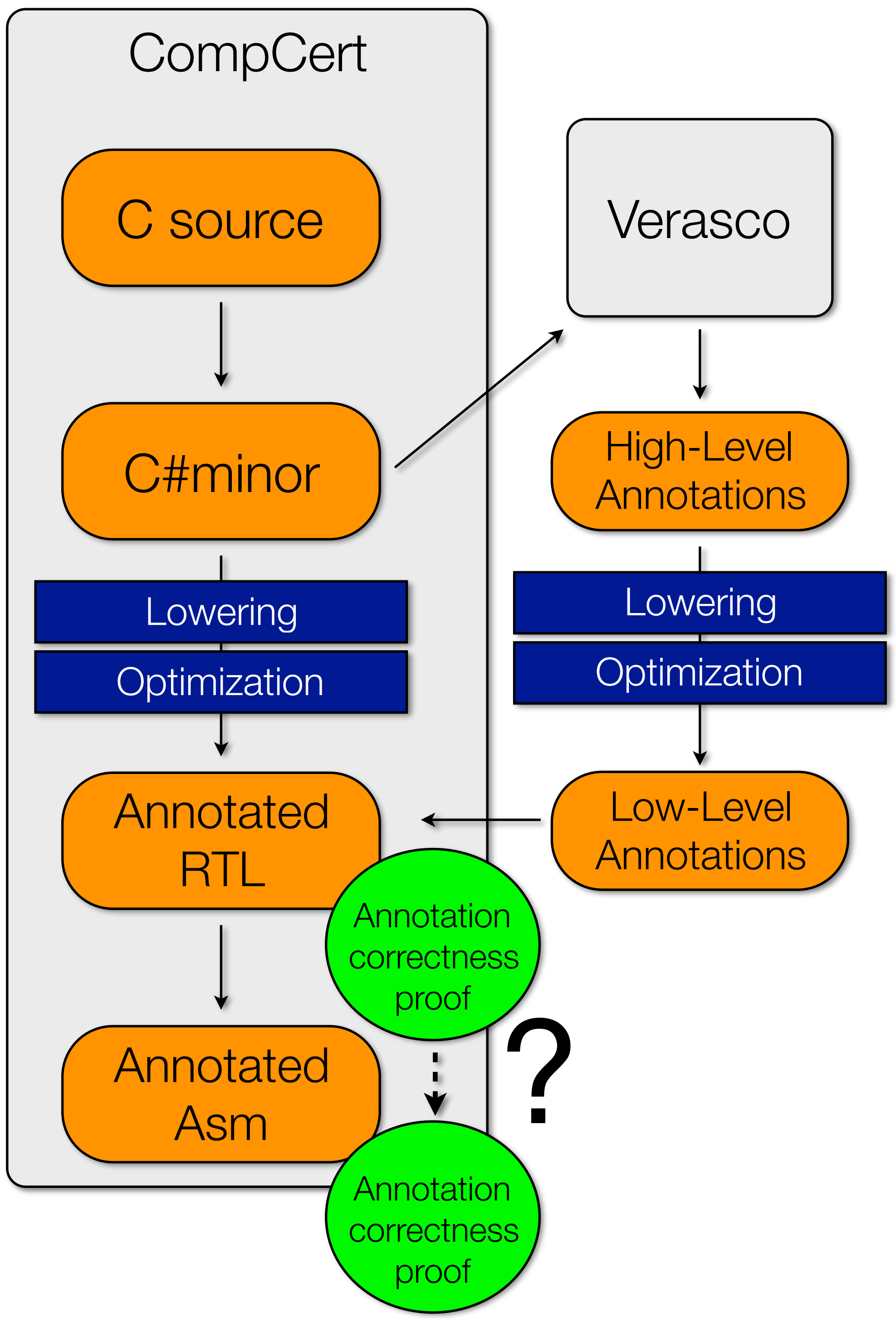


Translation validation of Verasco results

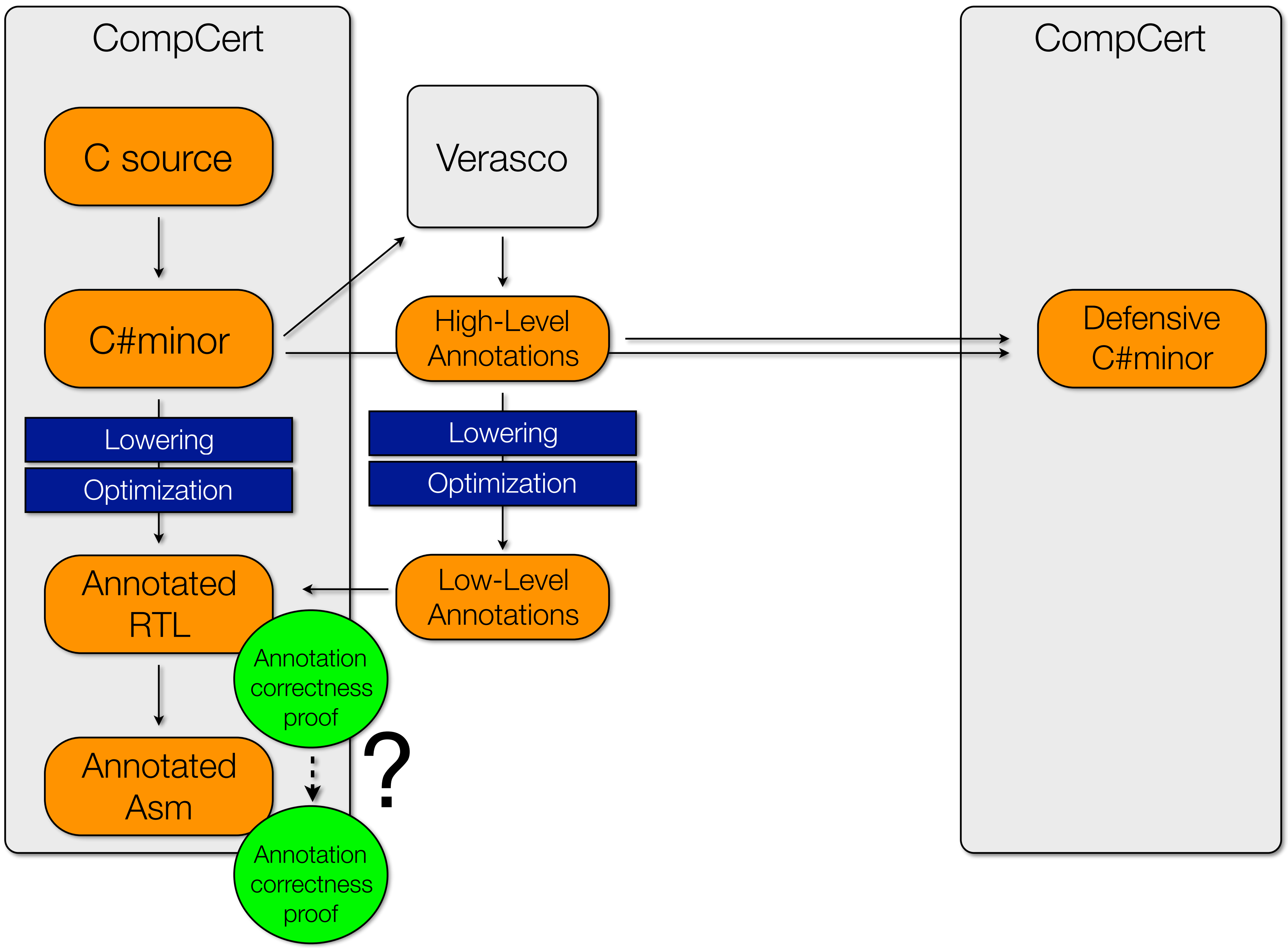
Translation validation of Verasco results



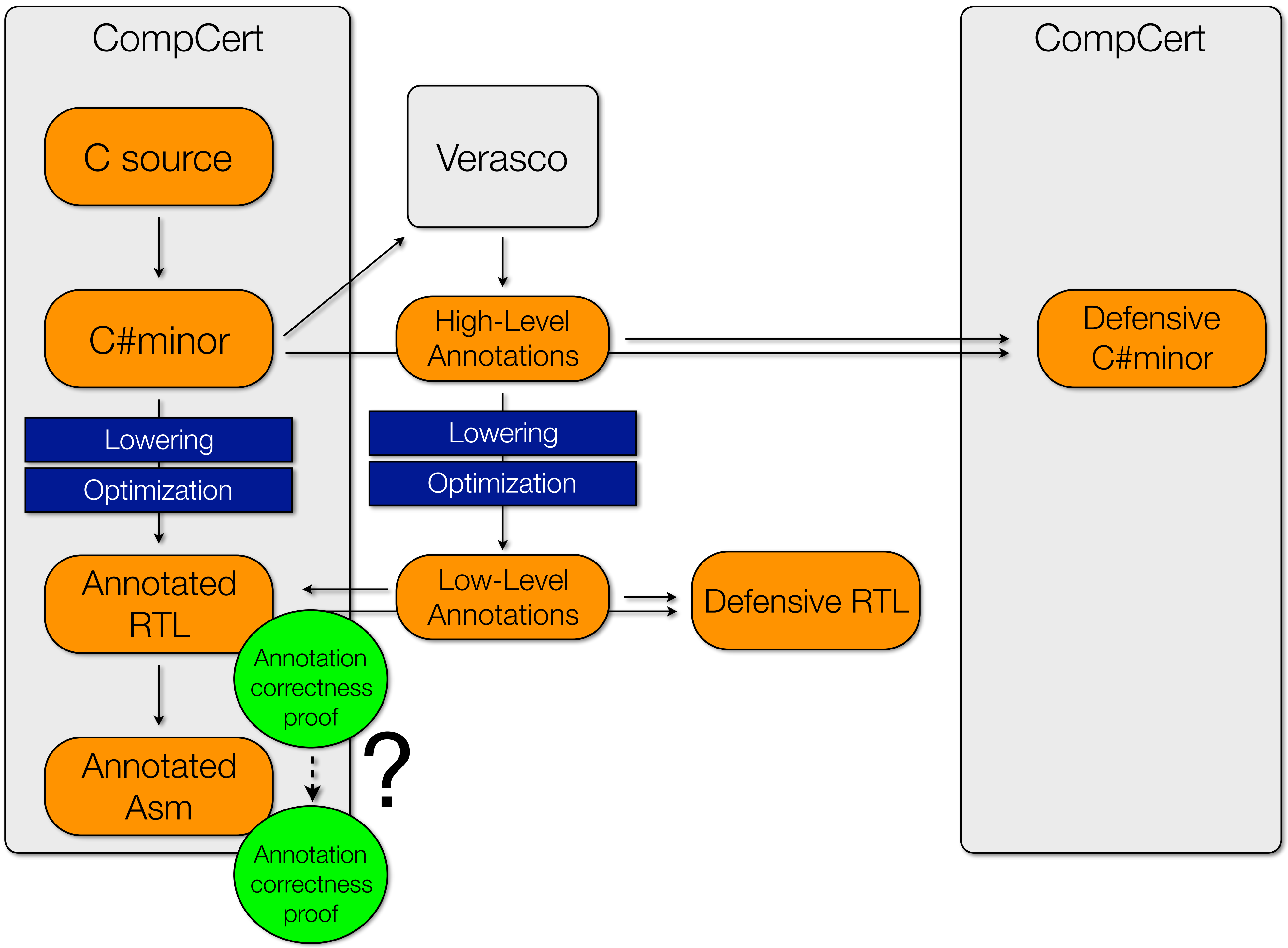
Translation validation of Verasco results



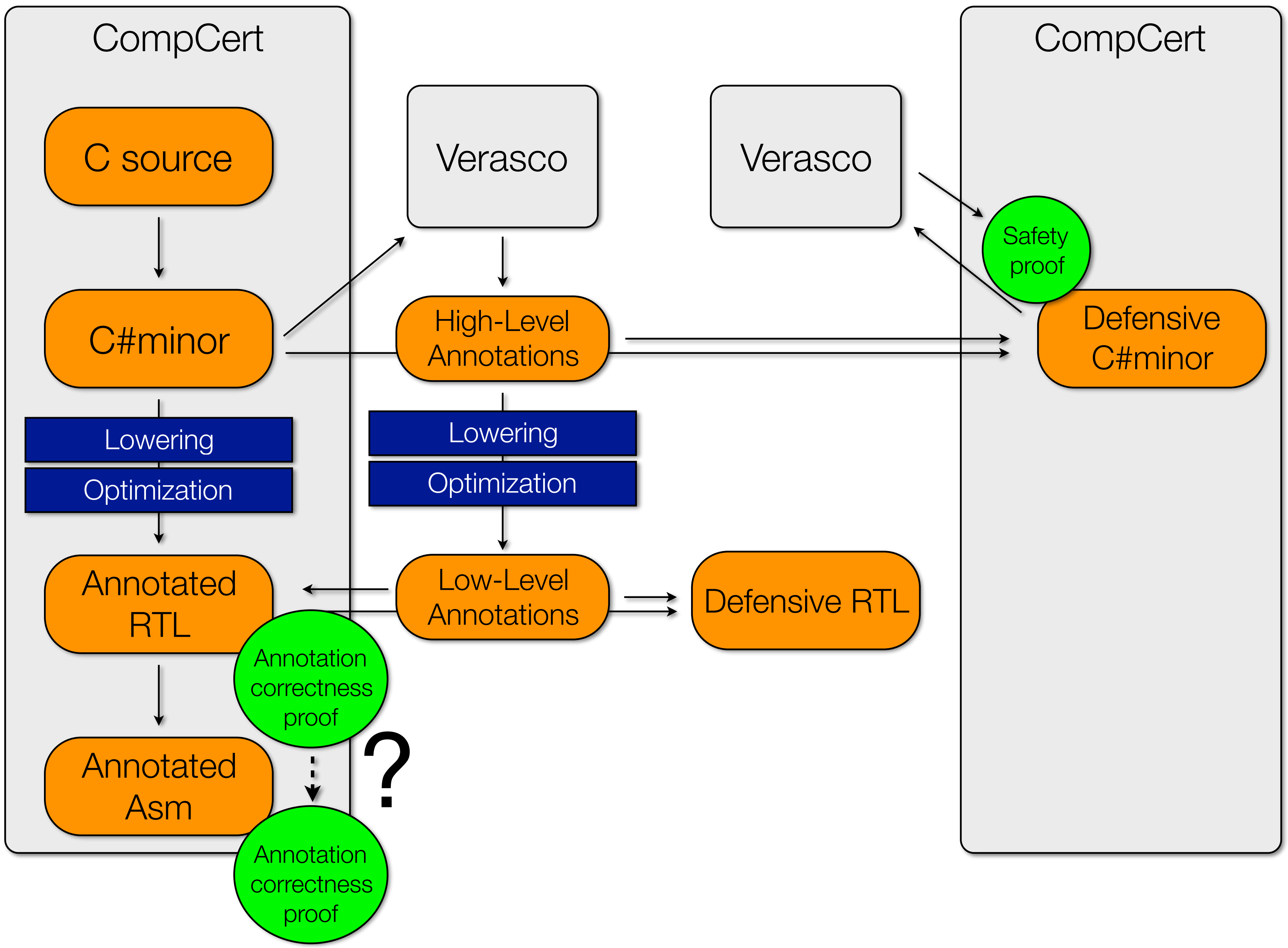
Translation validation of Verasco results



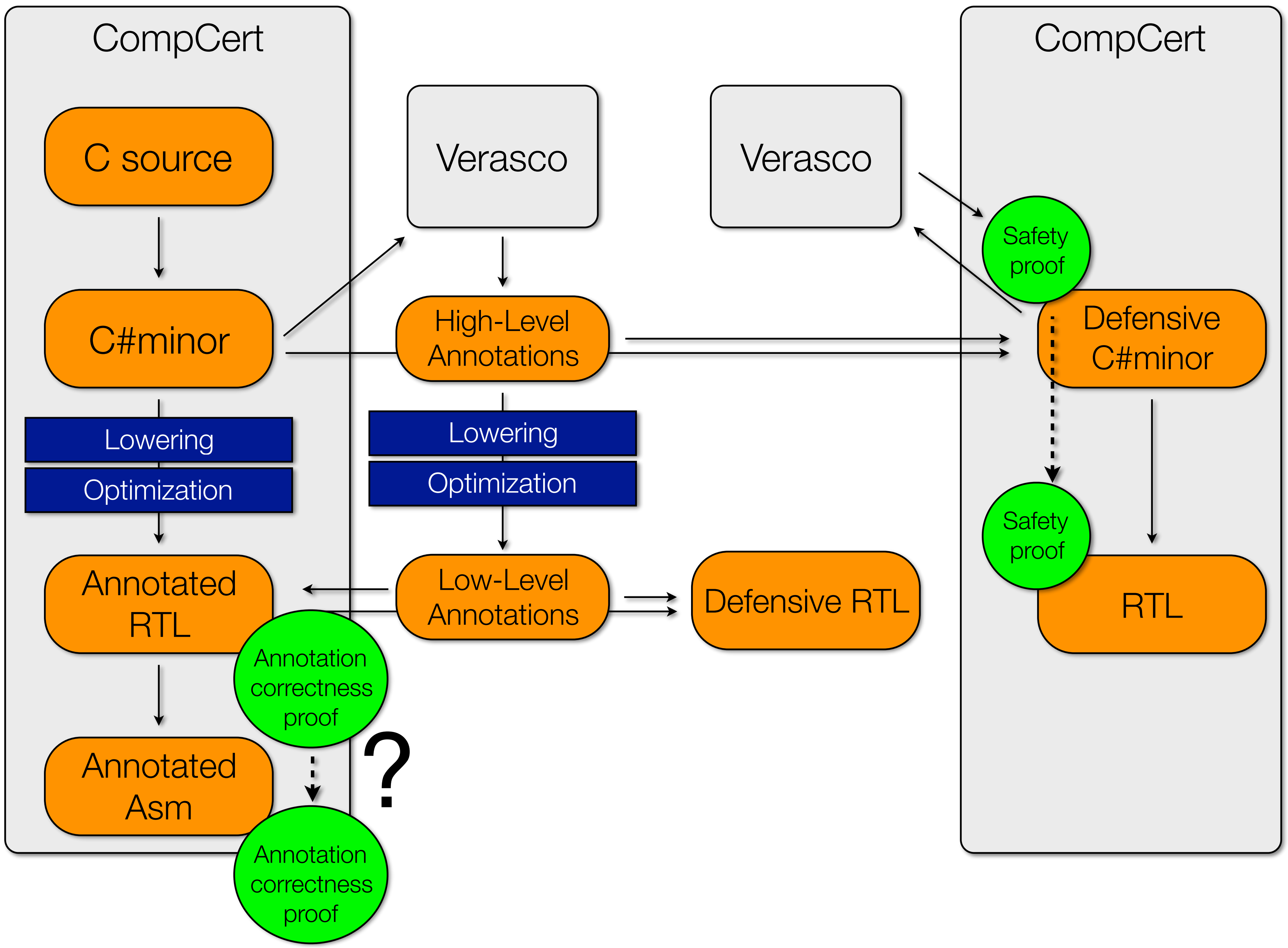
Translation validation of Verasco results



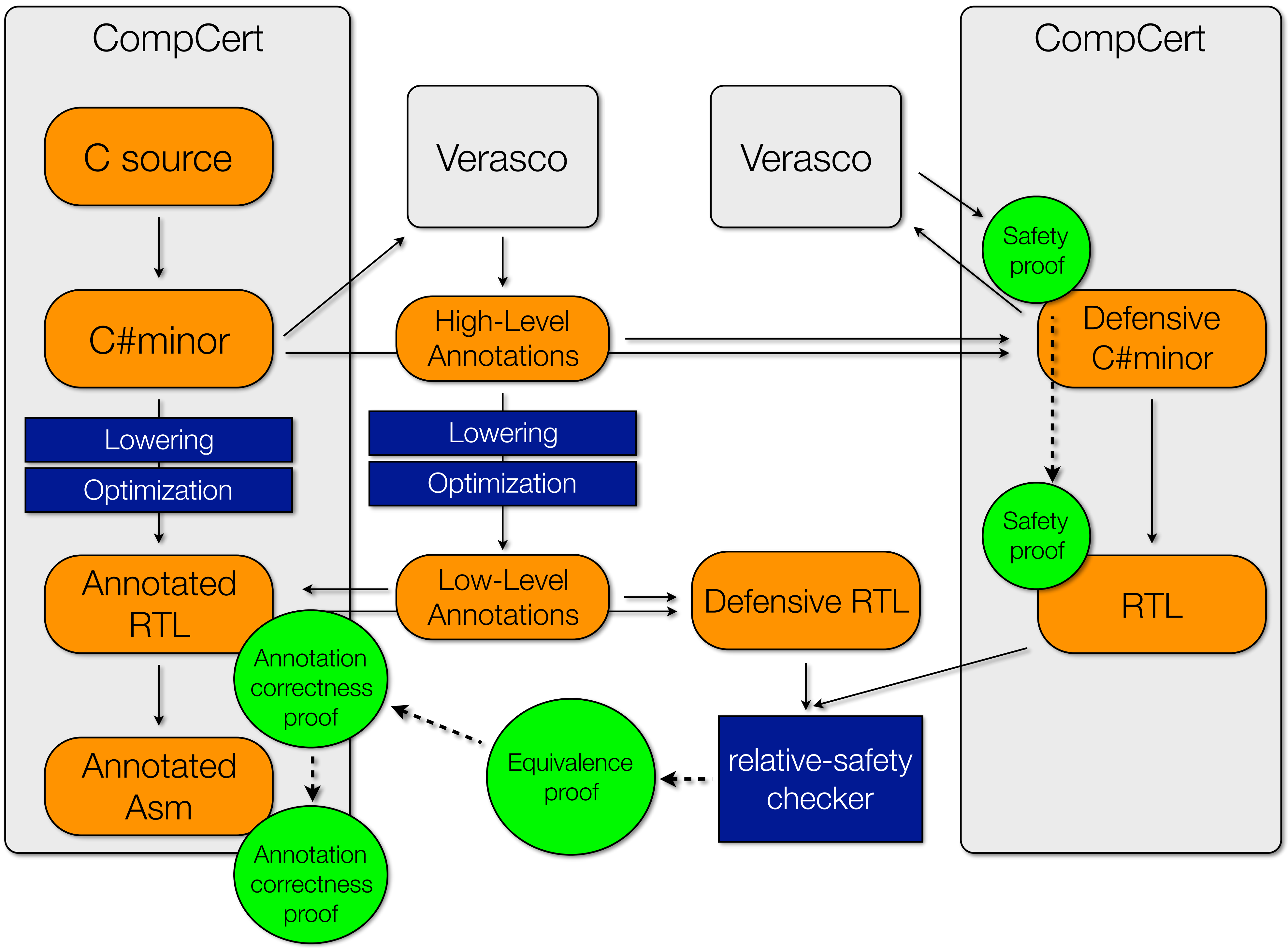
Translation validation of Verasco results



Translation validation of Verasco results



Translation validation of Verasco results



Third approach
Constant-time analysis at source level

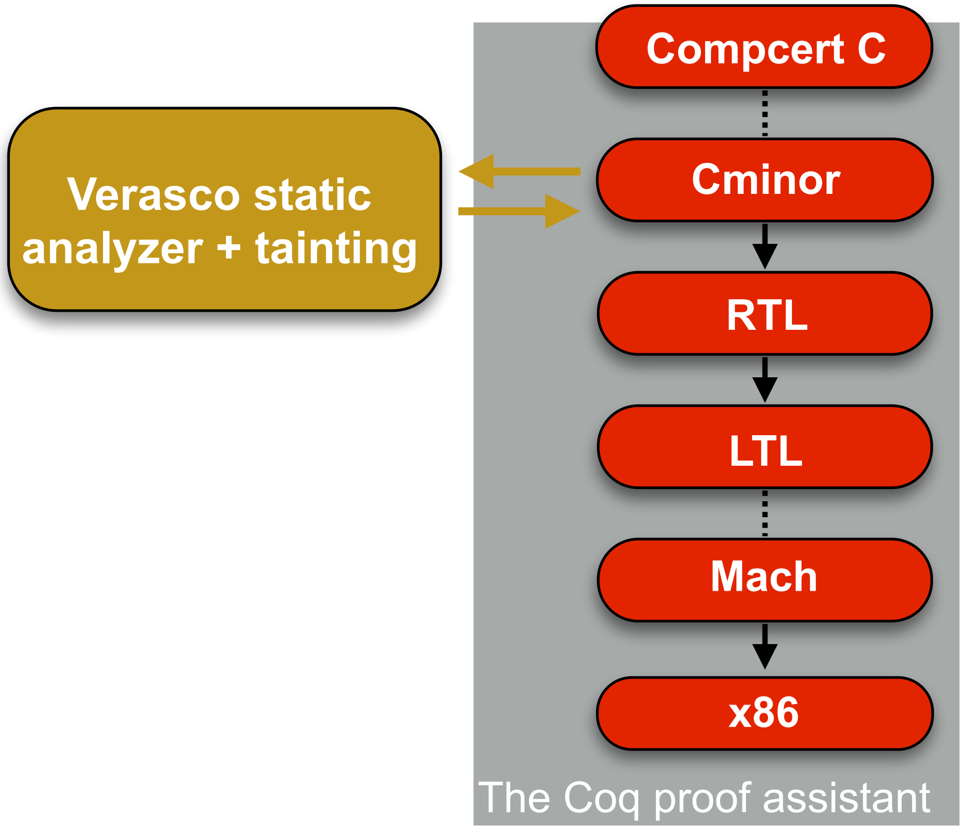
Sandrine Blazy, David Pichardie, Alix Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017.

Improvements

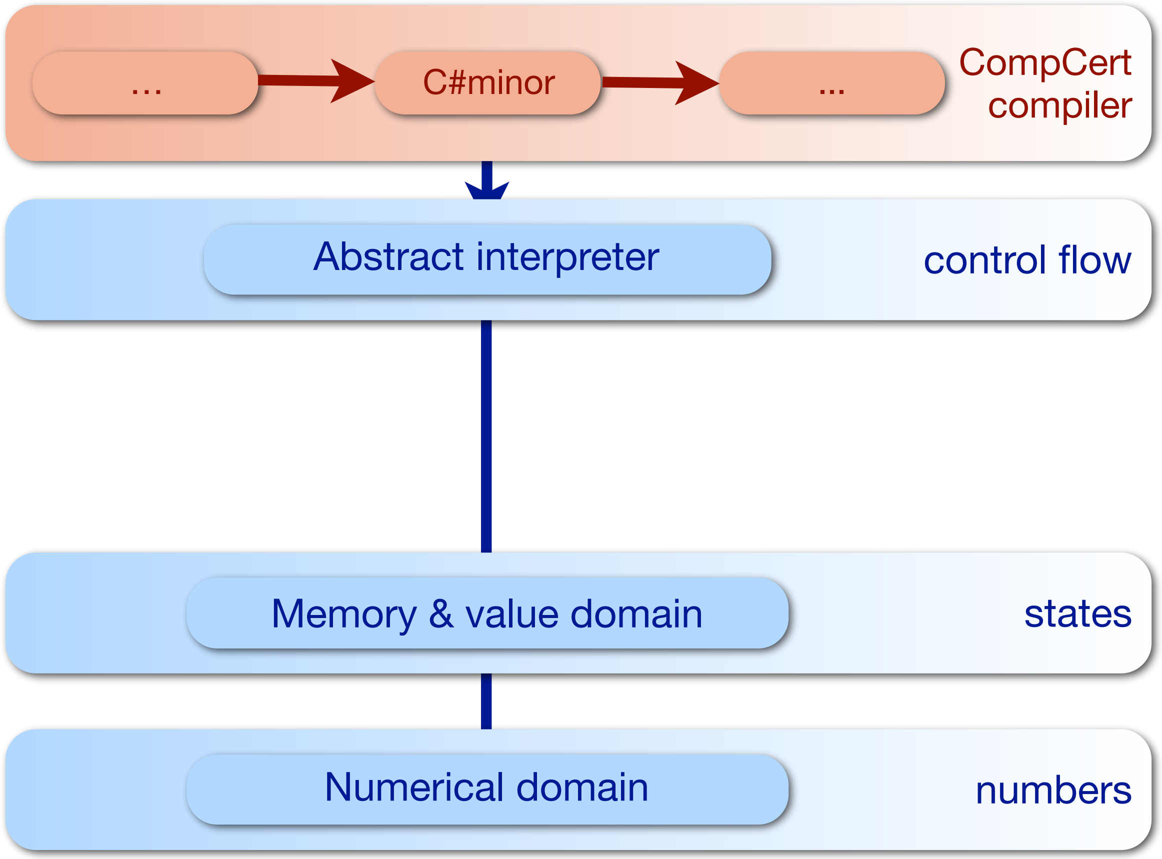
- Inform the programmer at source level
- Deeper interaction with Verasco

How?

- We mix Verasco memory abstract domain with fine-grained tainting

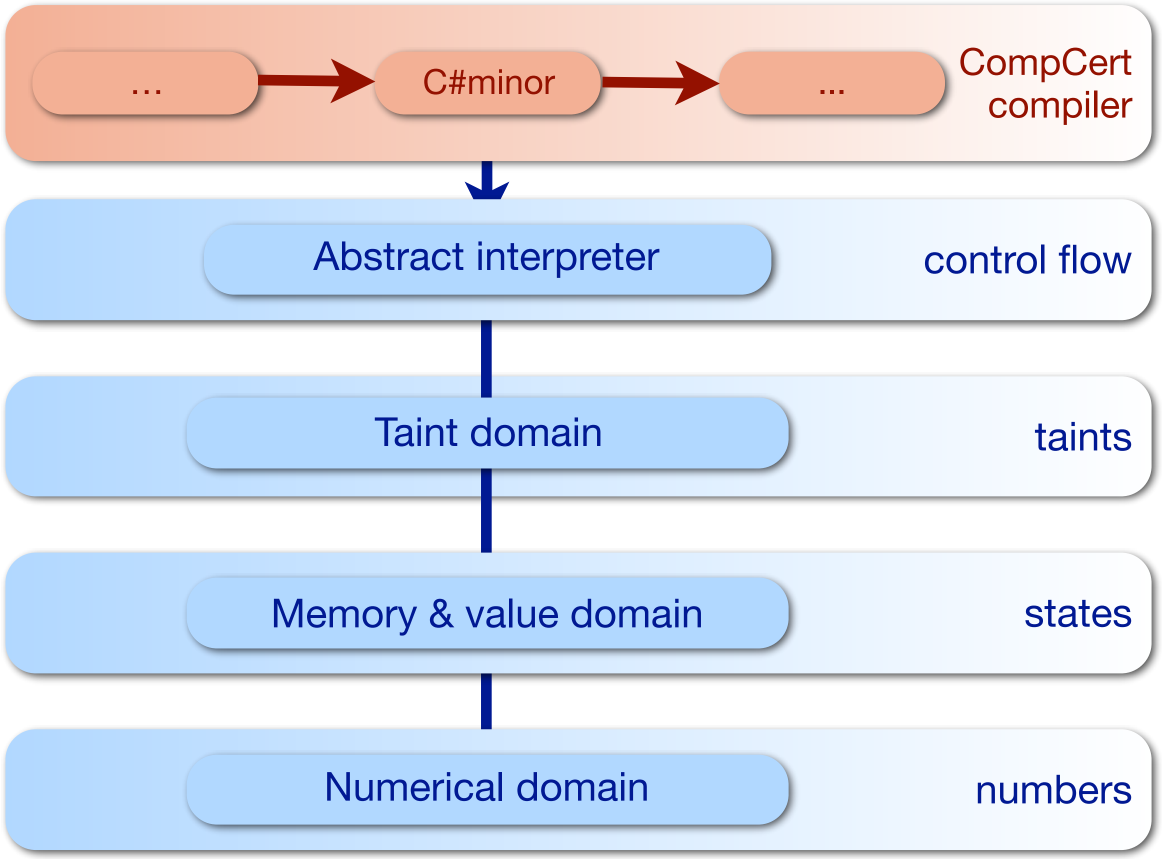


Constant-time analysis at source level



Constant-time analysis at source level

We design an *abstract functor*

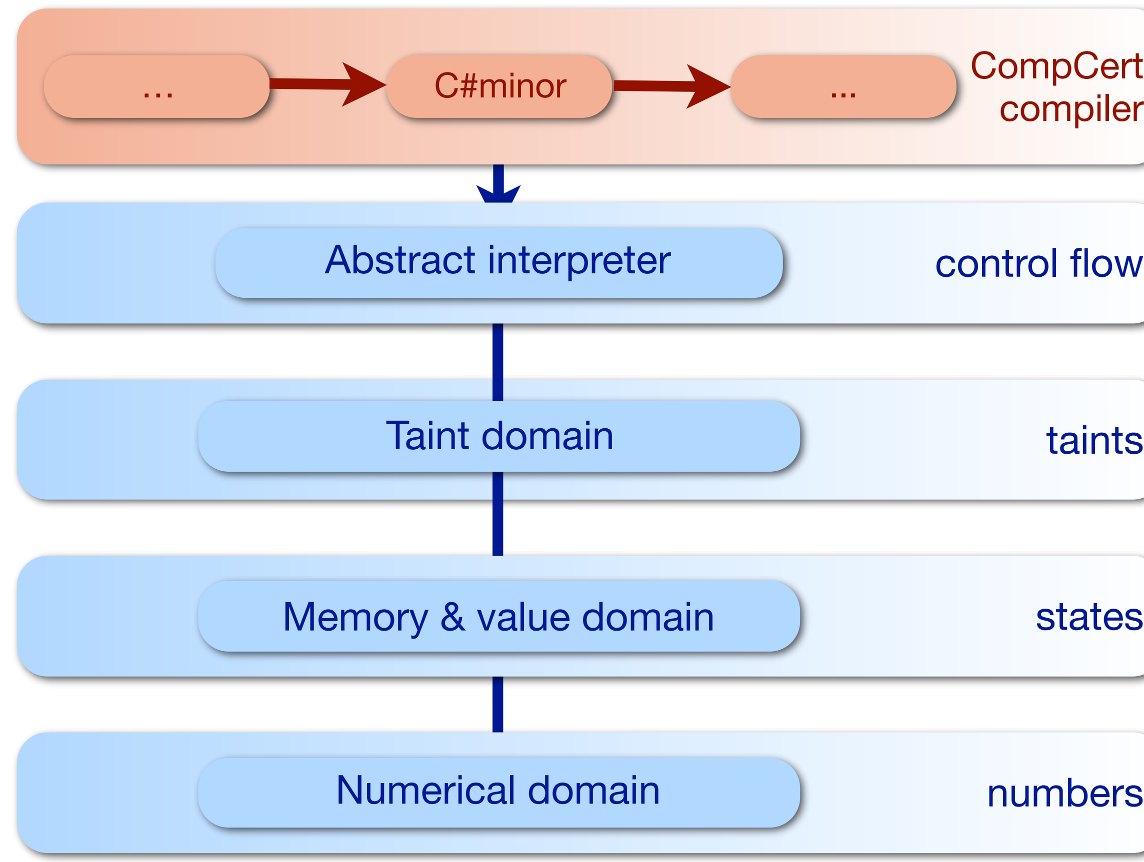


Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

$$
\begin{aligned}
[\![e]\!]^\sharp &: \quad \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \quad \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \quad \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \quad \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \quad \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \quad \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$



Constant-time analysis at source level

We design an *abstract functor*

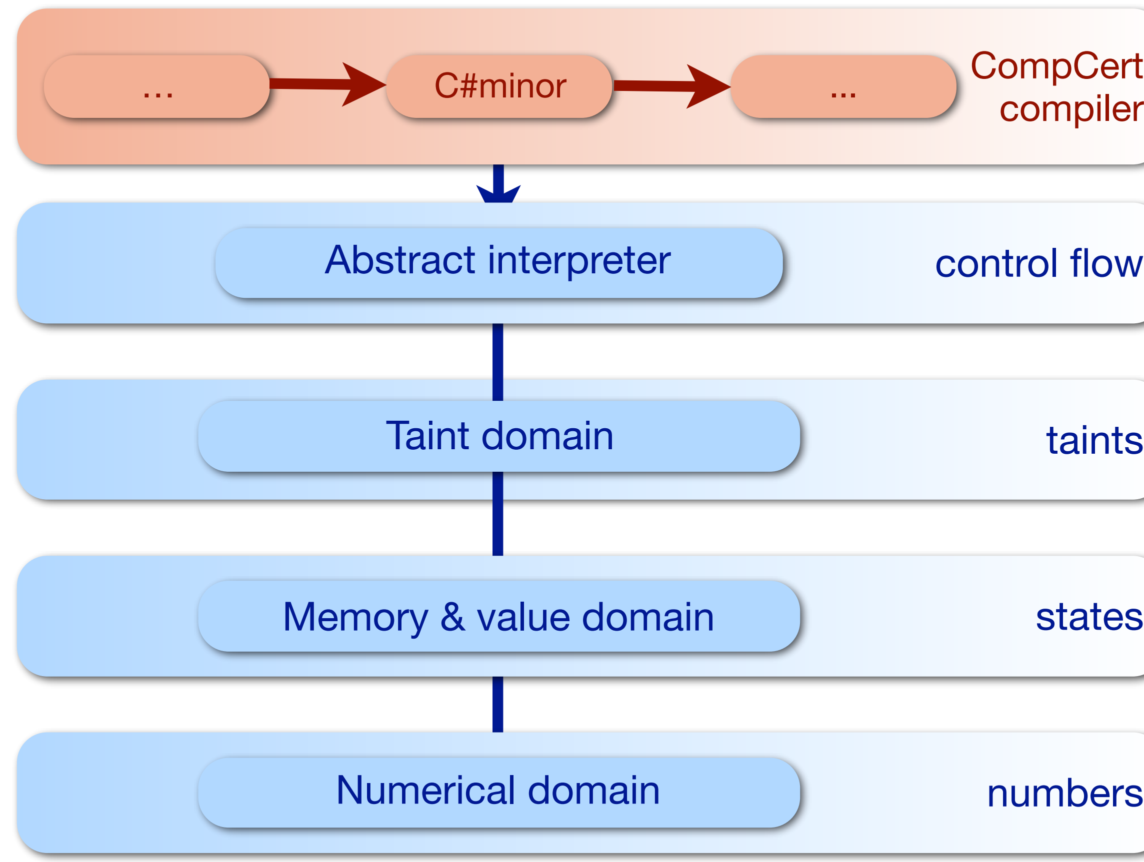
- takes as input an abstract memory domain

abstract memory

abstract value

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

set of concrete memory locations



Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

abstract memory

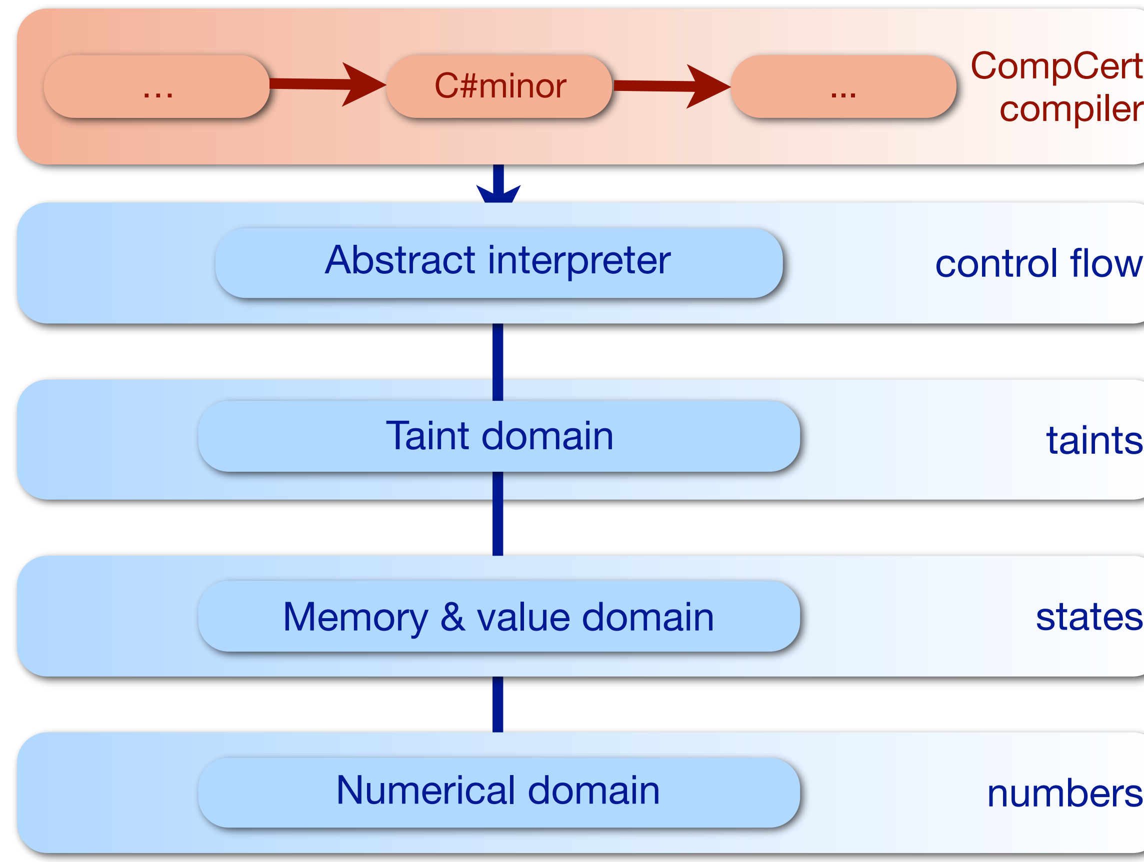
abstract value

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

set of concrete memory locations

- returns an abstract domain that taints every memory cells

$$
\begin{aligned}
\mathcal{T}[\![e]\!]^\sharp &: \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{V}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}}
\end{aligned}
$$



Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain

abstract memory

abstract value

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

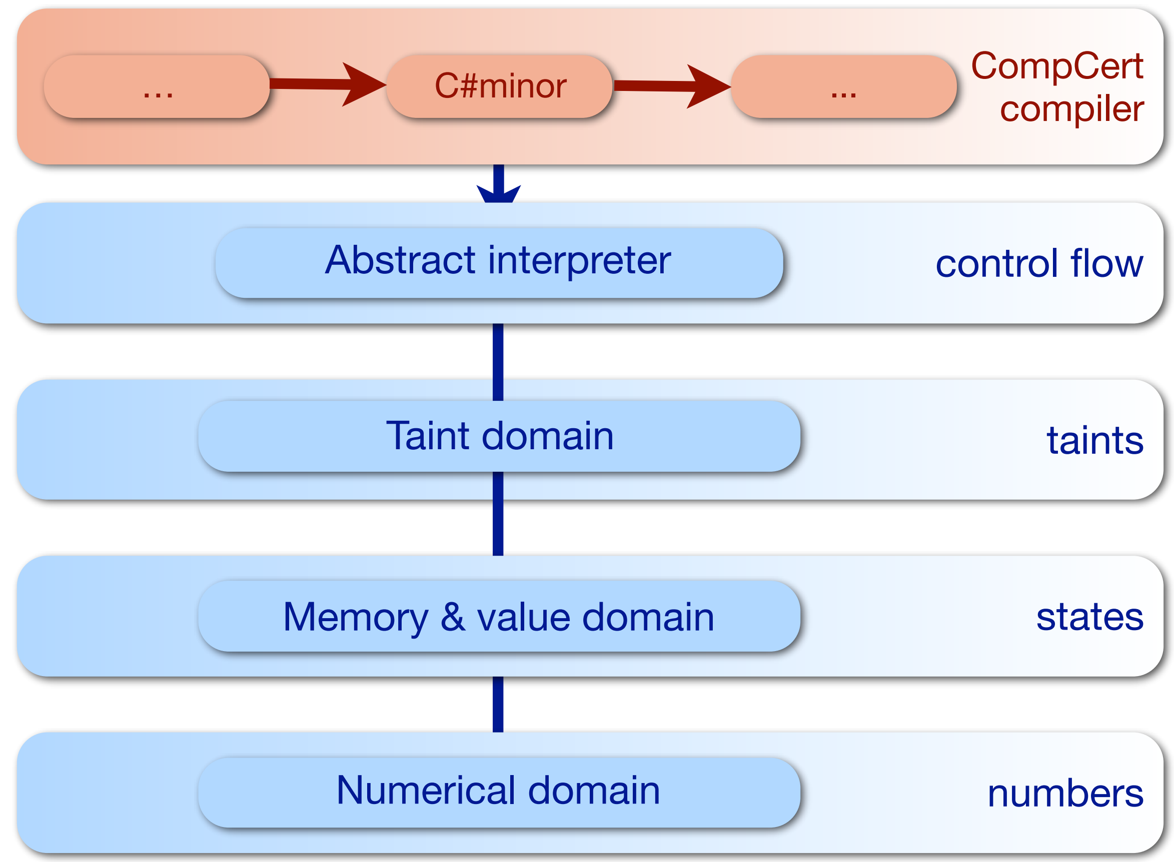
set of concrete memory locations

- returns an abstract domain that taints every memory cells

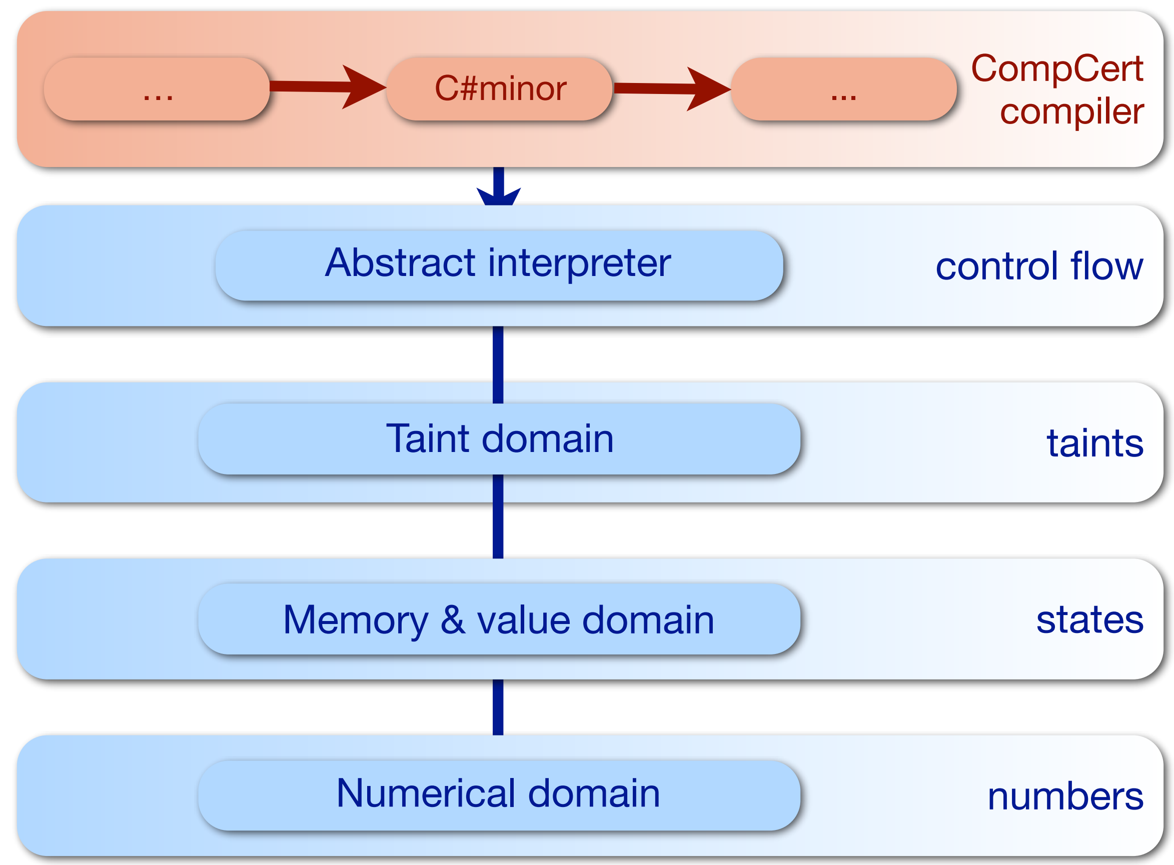
value taints {MustBeLow, MayBeHigh}

$$
\begin{aligned}
\mathcal{T}[\![e]\!]^\sharp &: \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{V}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}}
\end{aligned}
$$

tainting of each memory cell



Constant-time analysis at source level



We design an *abstract functor*

- takes as input an abstract memory domain

abstract memory

abstract value

$$
\begin{aligned}
[\![e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{assert}(e)^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathsf{concretize}^\sharp &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{L})
\end{aligned}
$$

set of concrete memory locations

- returns an abstract domain that taints every memory cells

value taints {MustBeLow, MayBeHigh}

$$
\begin{aligned}
\mathcal{T}[\![e]\!]^\sharp &: \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{V}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}} \\
\mathcal{T}[\![x \to *e]\!]^\sharp &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp_{\mathsf{taint}} \to \mathbb{M}^\sharp_{\mathsf{taint}}
\end{aligned}
$$

tainting of each memory cell

Example:

$$
\mathcal{T}[\![*e_1 \to e_2]\!]^\sharp(m^\sharp, t^\sharp) = t^\sharp\left[l \mapsto \mathcal{T}[\![e_2]\!]^\sharp\right]_{\forall l \in \mathsf{concretize}^\sharp \circ [\![e_1]\!]^\sharp(m^\sharp)}
$$

Experiments at source level (ESORICS'17)

Example	Size	Loc	Time
`aes`	1171	1399	41.39
`curve25519-donna`	1210	608	586.20
`des`	229	436	2.28
`rlwe_sample`	145	1142	30.76
`salsa20`	341	652	0.04
`sha3`	531	251	57.62
`snow`	871	460	3.37
`tea`	121	109	3.47
`nacl_chacha20`	384	307	0.34
`nacl_sha256`	368	287	0.04
`nacl_sha512`	437	314	1.02
`mbedtls_sha1`	544	354	0.19
`mbedtls_sha256`	346	346	0.38
`nbedtls_sha512`	310	399	0.26
`mee-cbc`	1959	939	933.37

Experiments at source level (ESORICS'17)

Example	Size	Loc	Time
aes	1171	1399	41.39
curve25519-donna	1210	608	586.20
des	229	436	2.28
rlwe_sample	145	1142	30.76
salsa20	341	652	0.04
sha3	531	251	57.62
snow	871	460	3.37
tea	121	109	3.47
nacl_chacha20	384	307	0.34
nacl_sha256	368	287	0.04
nacl_sha512	437	314	1.02
mbedtls_sha1	544	354	0.19
mbedtls_sha256	346	346	0.38
nbedtls_sha512	310	399	0.26
mee-cbc	1959	939	933.37

Same benchmarks than Almeida et al.

J.B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir and M.Emmi. *Verifying Constant-Time Implementations.* USENIX Security Symposium 2016.

Not handled by Almeida et al. because LLVM alias analysis limitations

Comparing the three approaches

Approach	Pro	Cons	Current state of proof mechanization
Direct analysis at pre-assembly level	property established at the expected level	engineering a static analysis at assembly level is hard	fully verified in Coq
Translation of Verasco results	the translation mechanism may be useful outside security analysis	the validation technique may be incomplete with respect to state-of-the-art compiler optimizations	only the annotation validation is currently verified
Analysis at source level	1) reuse the Verasco effort 2) feedback for crypto programmers	we need to trust (or prove) that the compiler will not break the security property	only a paper proof

Conclusions

Constant-time

- simpler than full non-interference but still challenging security property
- hard to obtain at assembly level without control on the compiler
- further work: cover more side-channels (e.g. floating point computations)

Verified C compiler toolchain for security

- strong soundness guaranties
- allow experimentation with real crypto programs
- further work: enforce other folklore protections