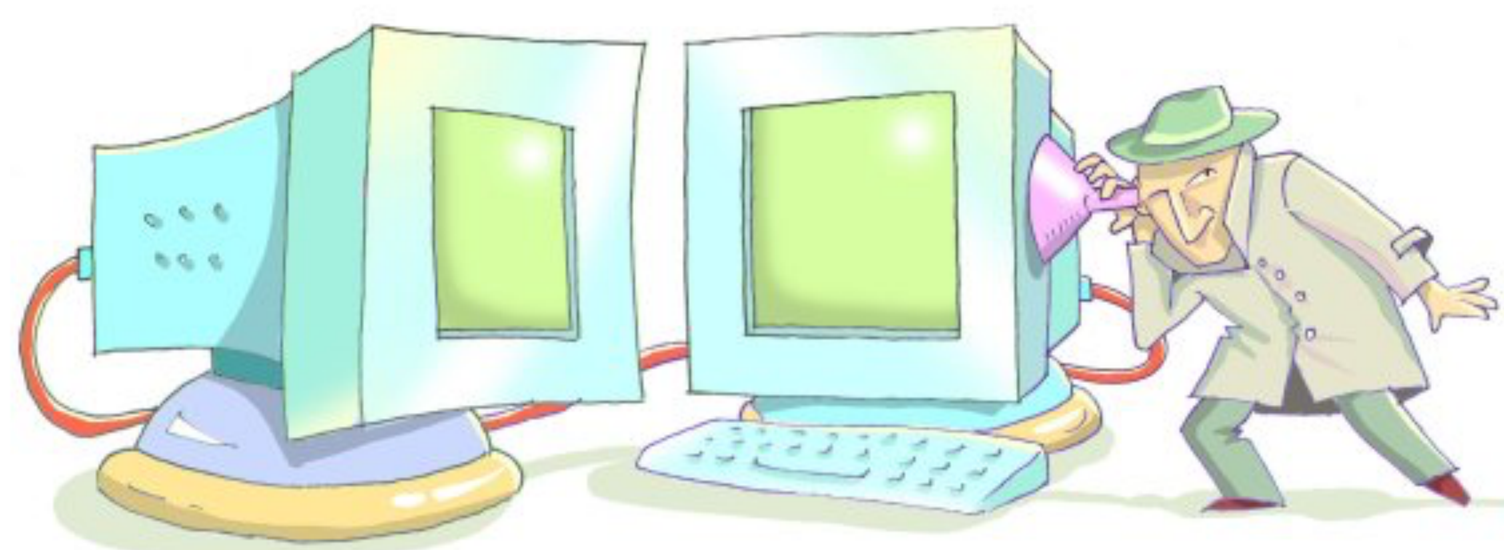
Verification of constant-time implementation in a verified compiler toolchain

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

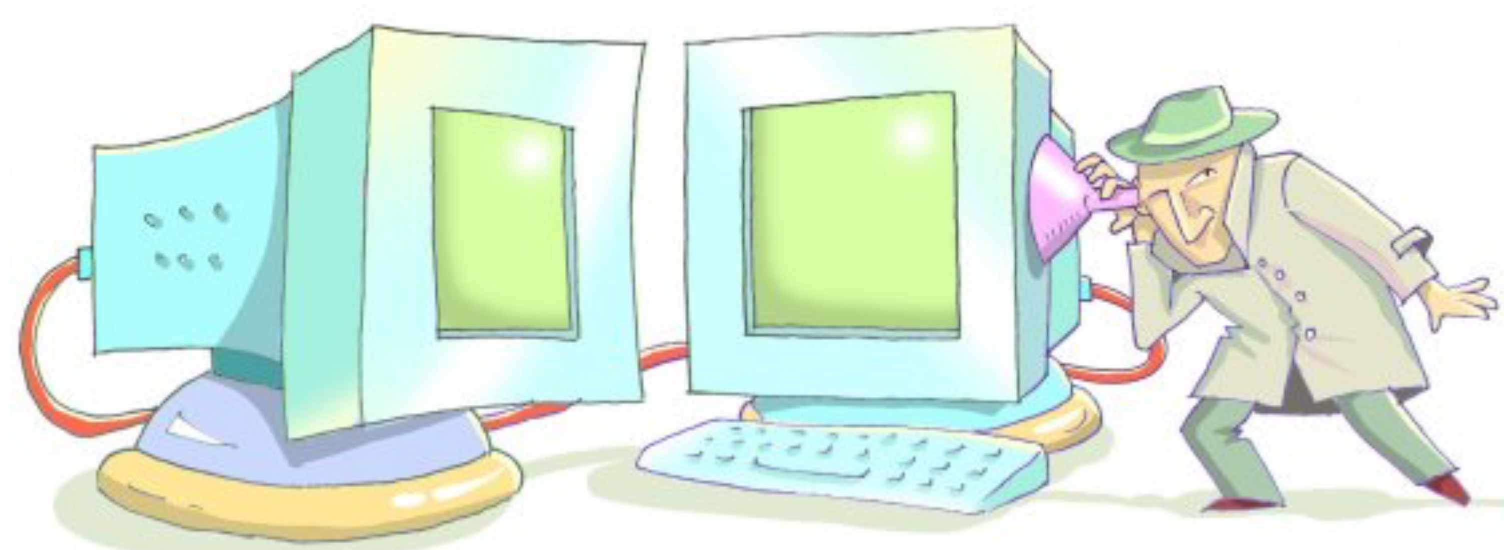


Constant-time programs

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



Constant-time programs

Example

Constant-time programs

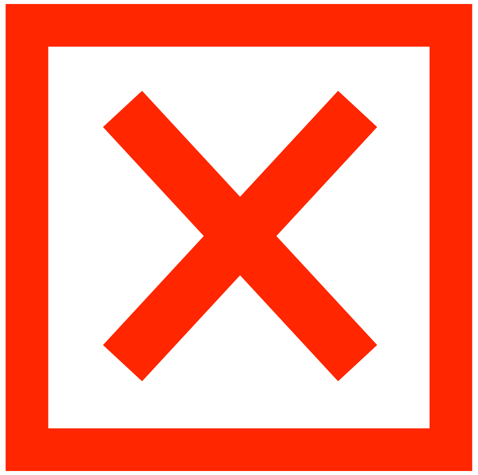
Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```

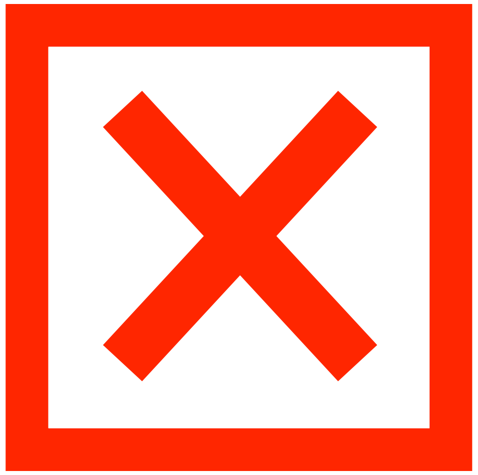


Not constant-time

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```



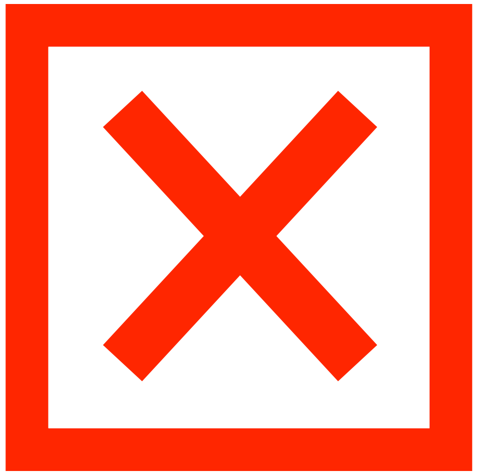
Not constant-time

```
boolean testPIN(int code[]) {
  int diff = 0;
  for (int i=0; i<N; i++) {
    diff = diff | (code[i] ^ secret[i]);
  }
  return (diff == 0);
}
```

Constant-time programs

Example

```
boolean testPIN(int code[]) {
  for (int i=0; i<N; i++) {
    if (code[i] != secret[i]) return false;
  }
  return true;
}
```



Not constant-time

```
boolean testPIN(int code[]) {
  int diff = 0;
  for (int i=0; i<N; i++) {
    diff = diff | (code[i] ^ secret[i]);
  }
  return (diff == 0);
}
```

Constant-time

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

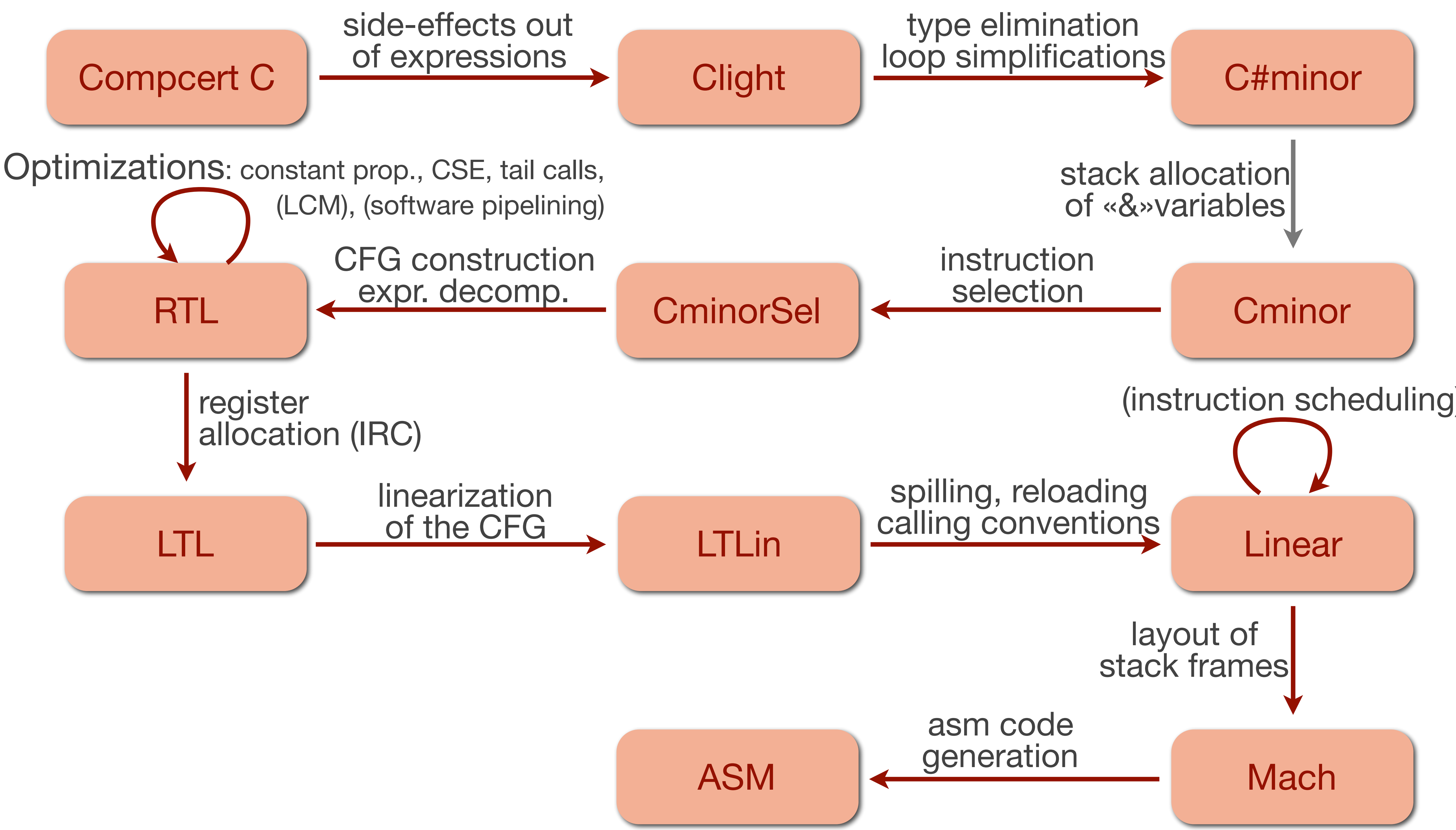
= compiler + proof that the compiler does not introduce bugs

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

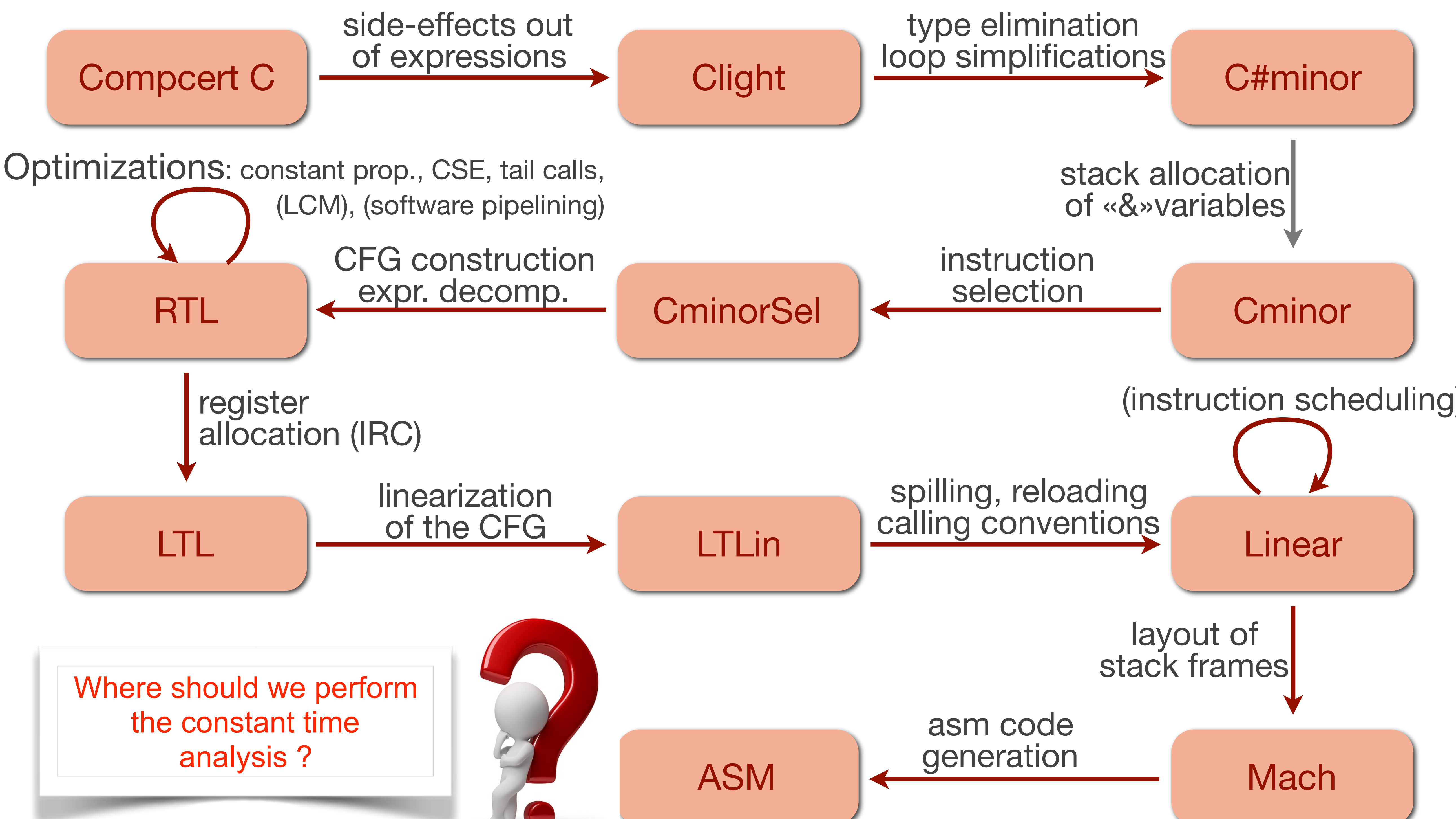
> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> then «C behaves like S».

does not deal with the constant-time security property !

CompCert: 1 compiler, 11 languages



CompCert: 1 compiler, 11 languages



Where should we perform the constant time analysis ?

Our approach

1. Analyse the program at source level

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

2. Make the compiler preserve the property

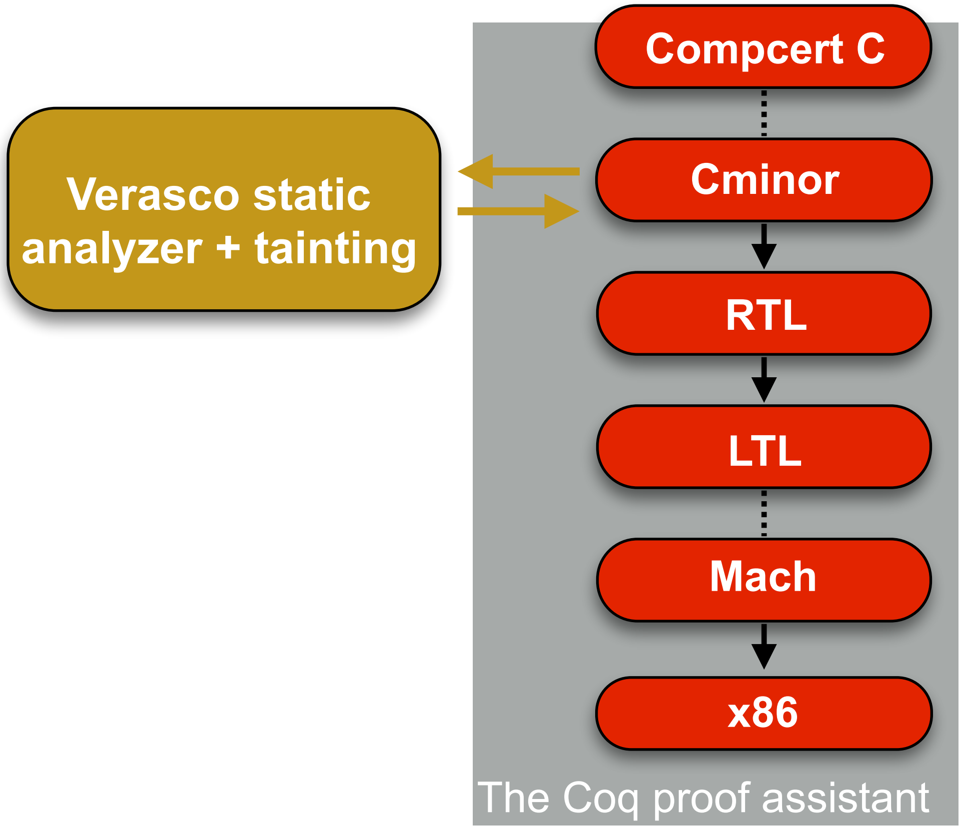
G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu.
Formal verification of a constant-time preserving C compiler.
POPL 2020.

Constant-time analysis at source level

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

We perform static analysis at (almost) C level

- Based on previous work with a value analyser, Verasco
- We mix Verasco memory tracking with fine-grained tainting



The Verasco project

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

http://verasco.imag.fr

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

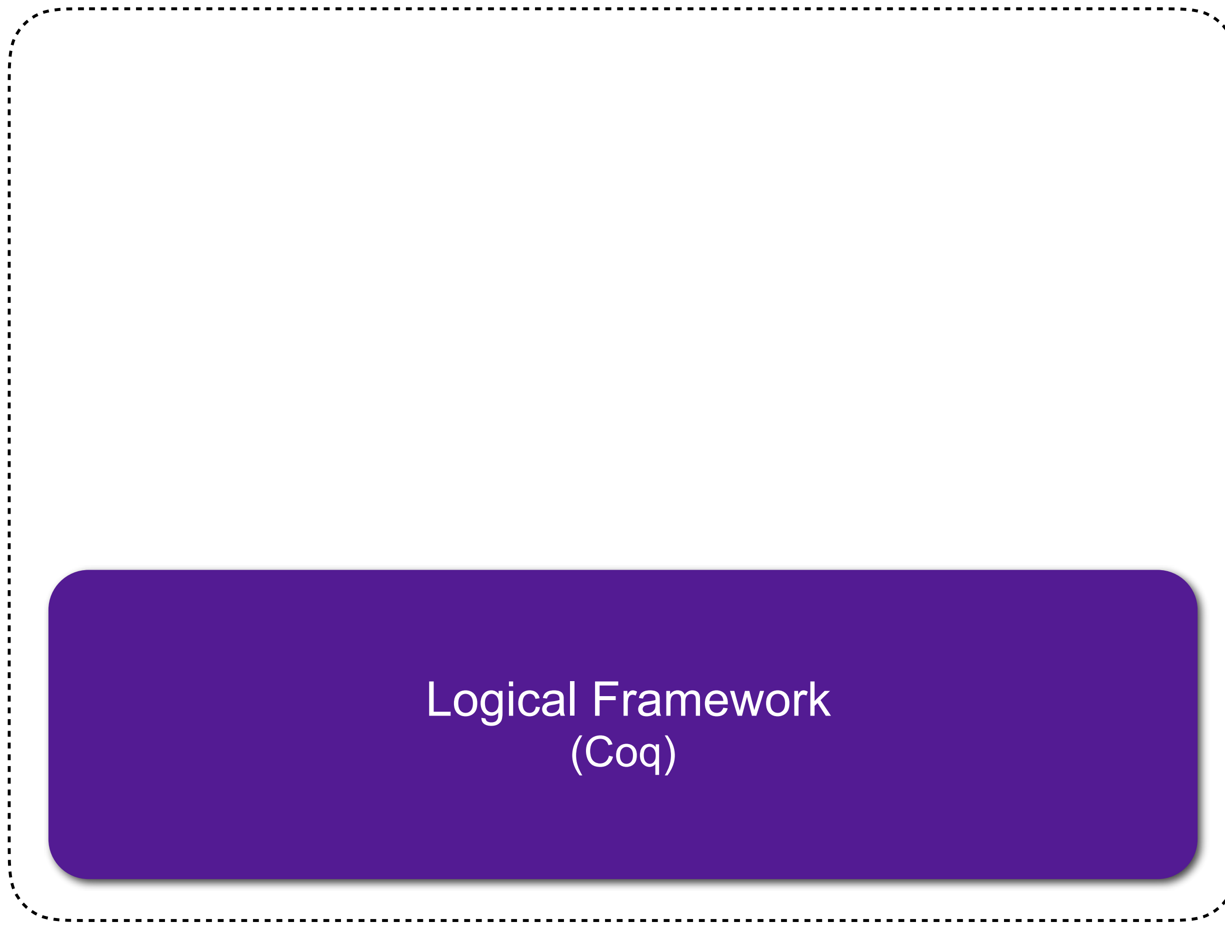
- Language analyzed: the CompCert subset of C
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée's
- To prove the absence of undefined behaviors in C source programs

Slogan:

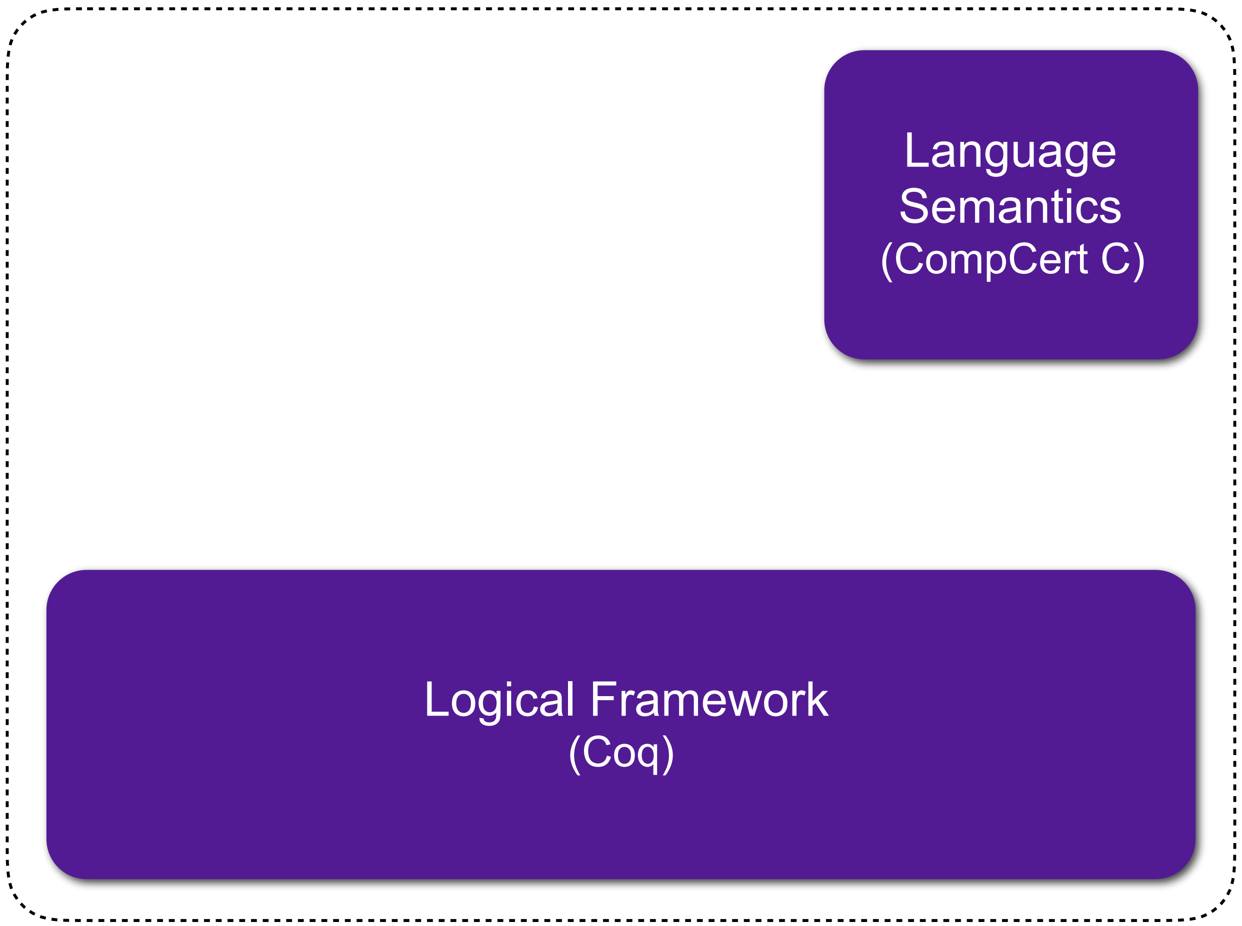
- if « CompCert ≈ 1/10th of GCC but formally verified »,
- likewise « Verasco ≈1/10th of Astrée but formally verified »

Verified Static Analysis

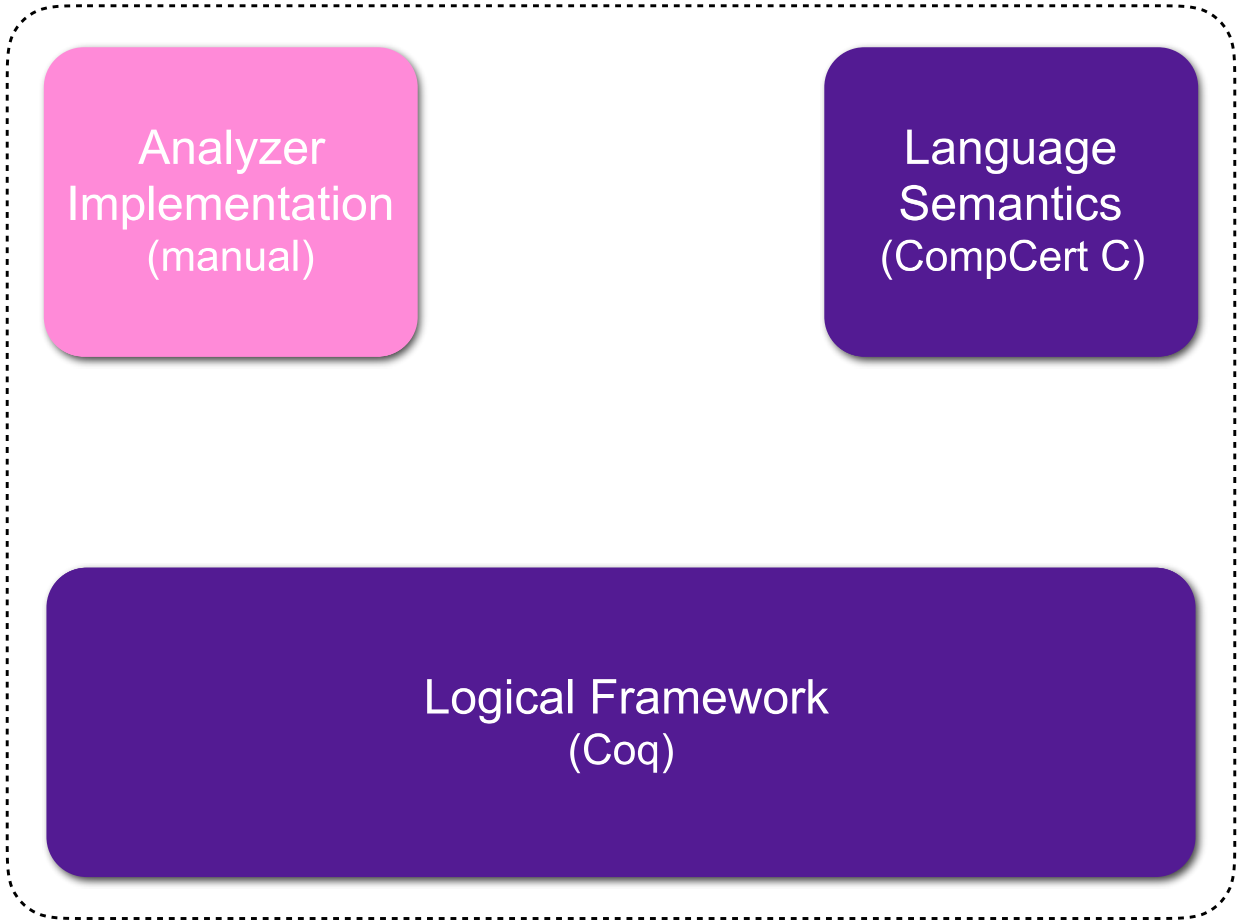
Verified Static Analysis



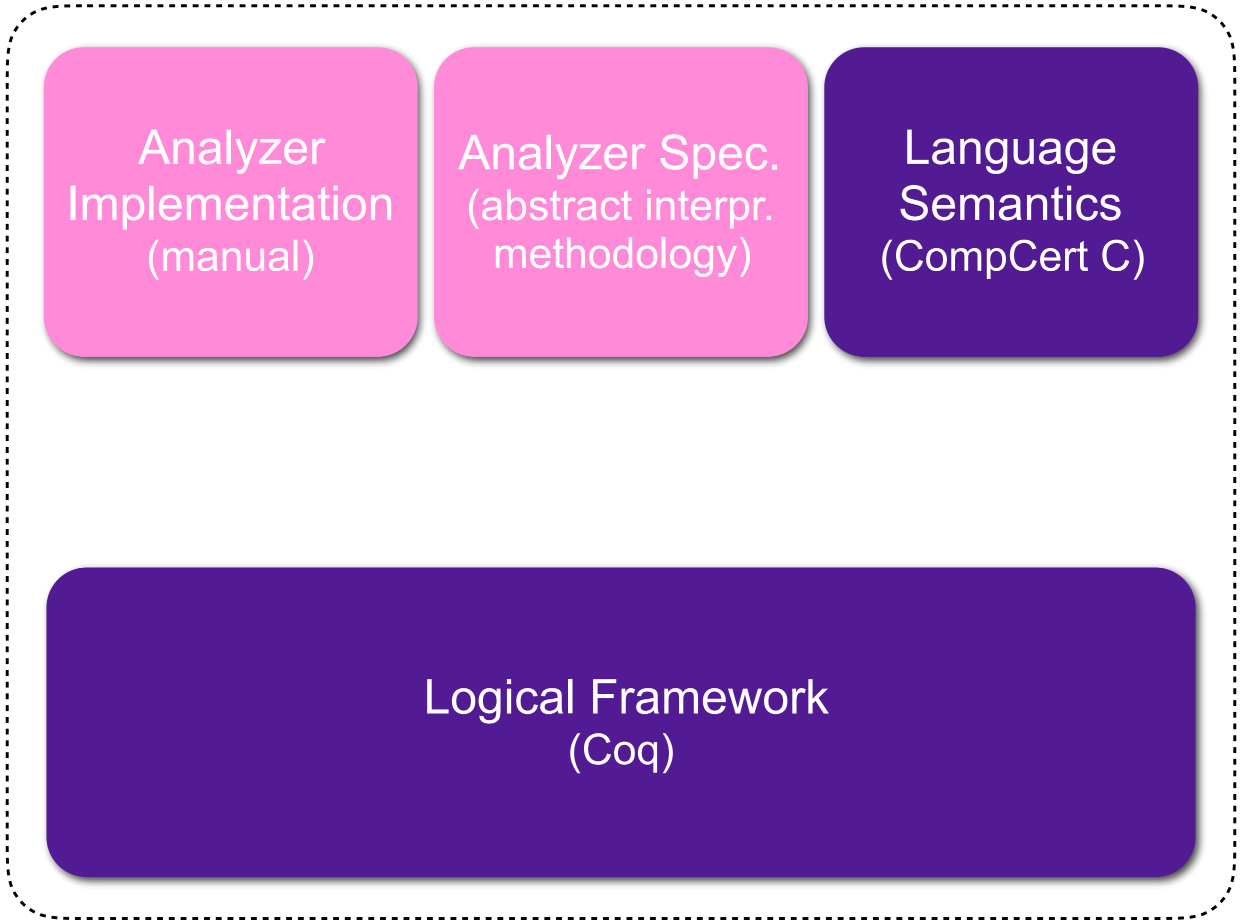
Verified Static Analysis



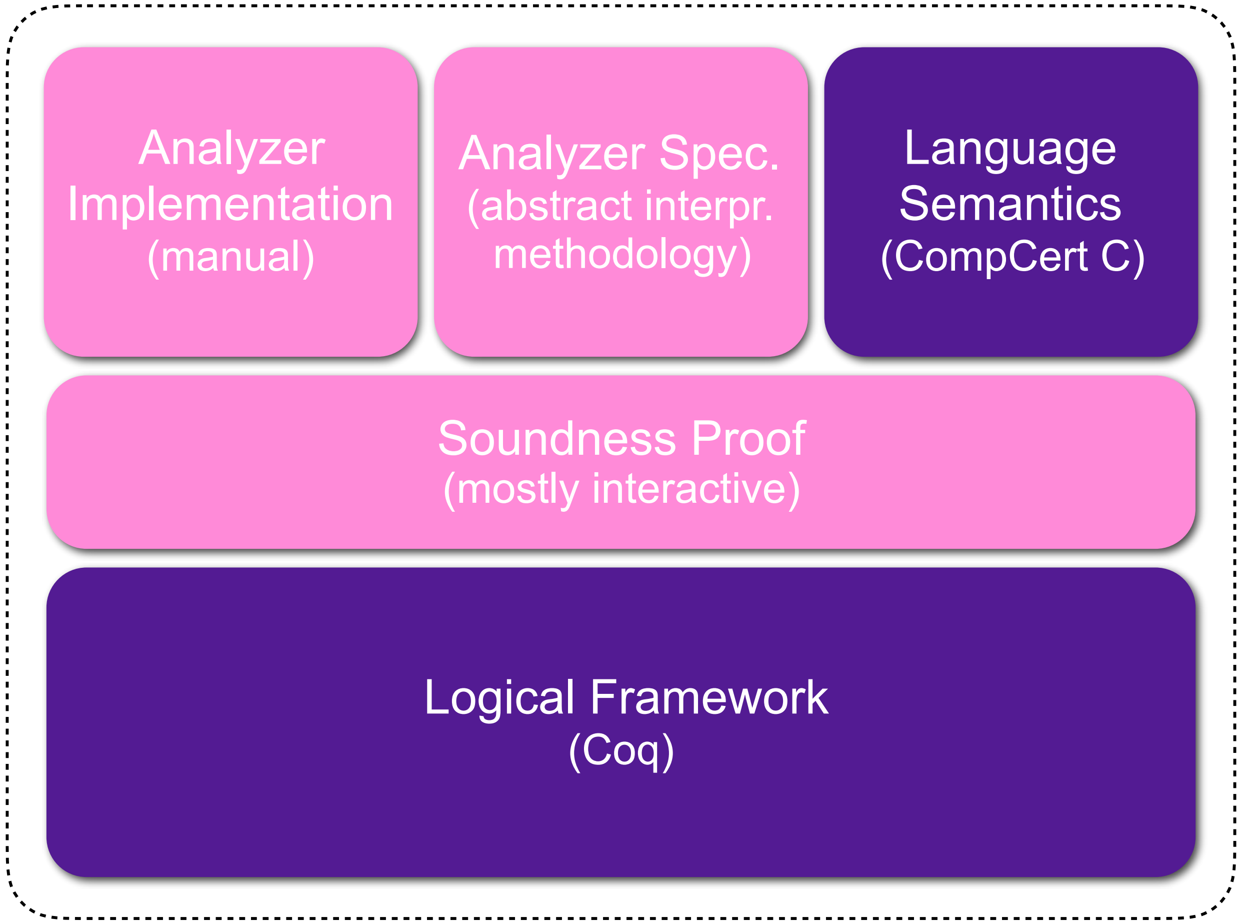
Verified Static Analysis



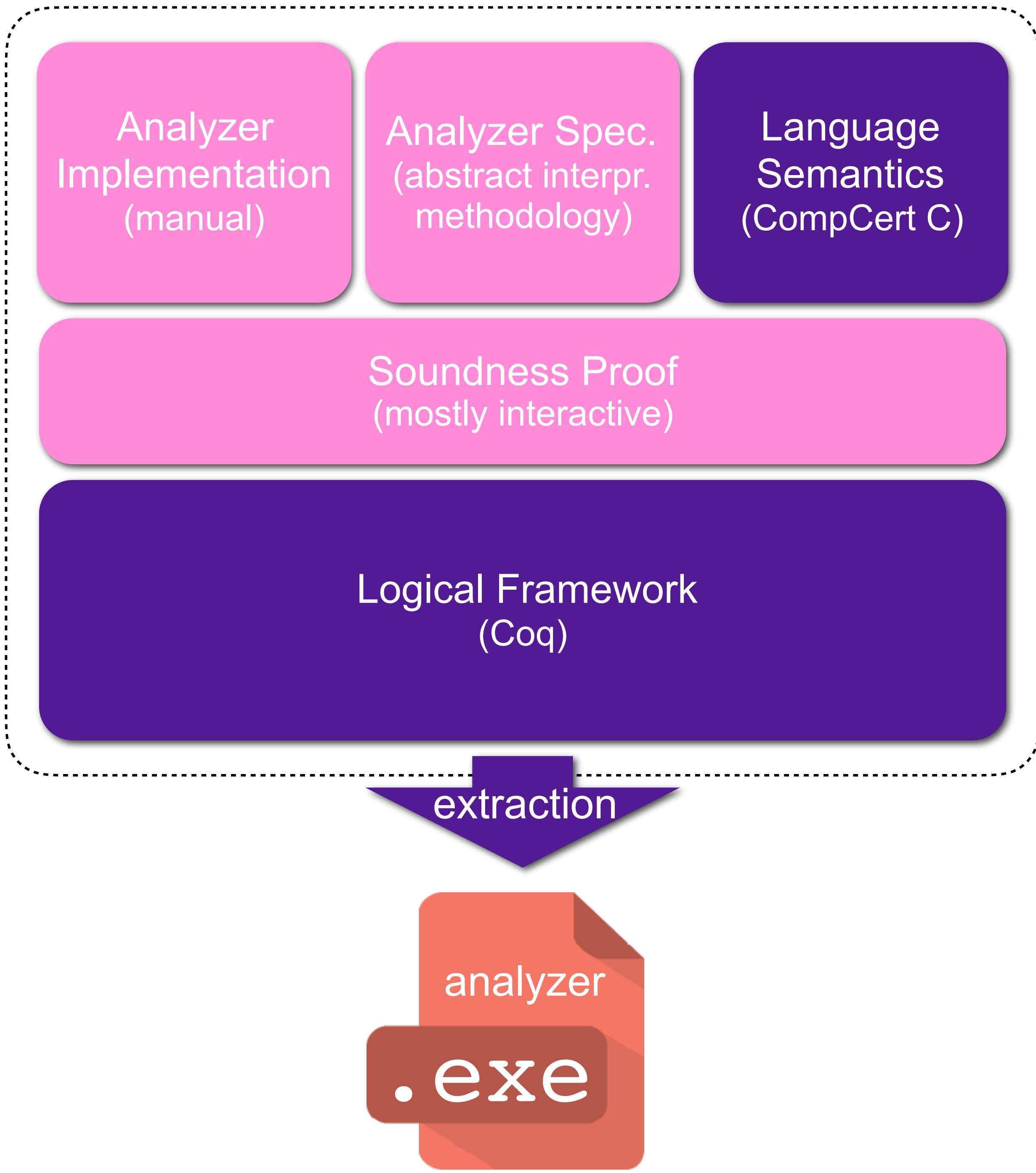
Verified Static Analysis



Verified Static Analysis



Verified Static Analysis

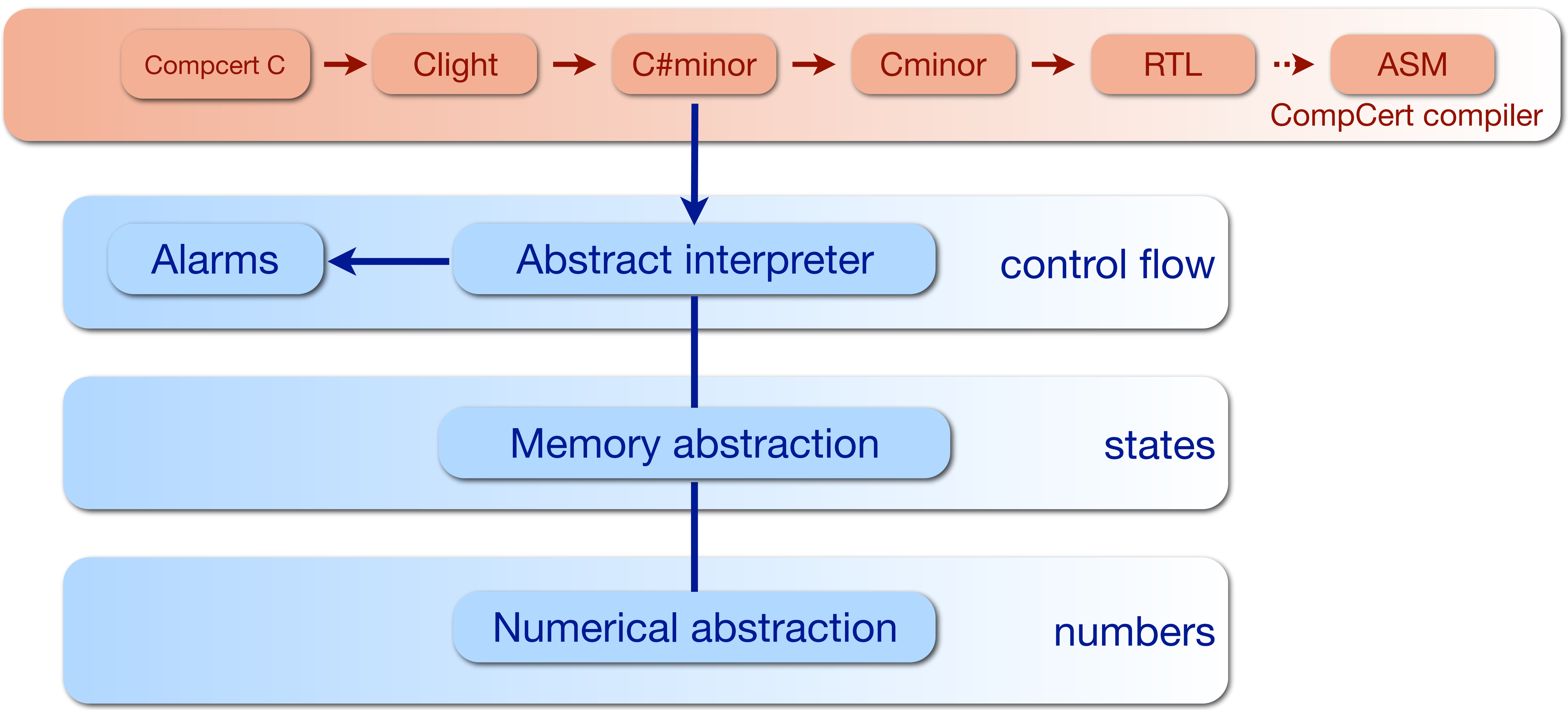


Verasco

A Formally-Verified C Static Analyzer

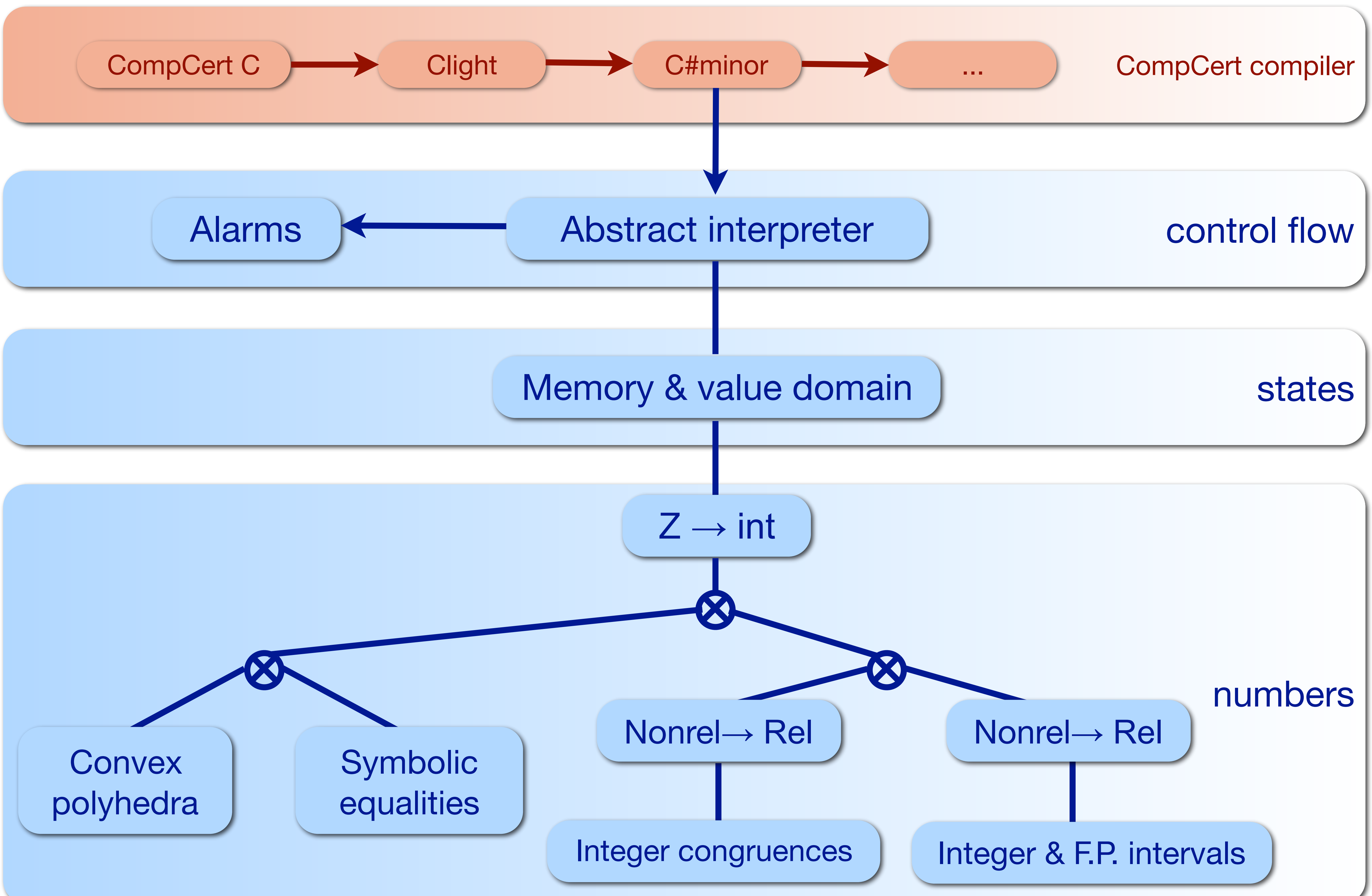
JH. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie.
A Formally-Verified C Static Analyzer.
POPL 2015.

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



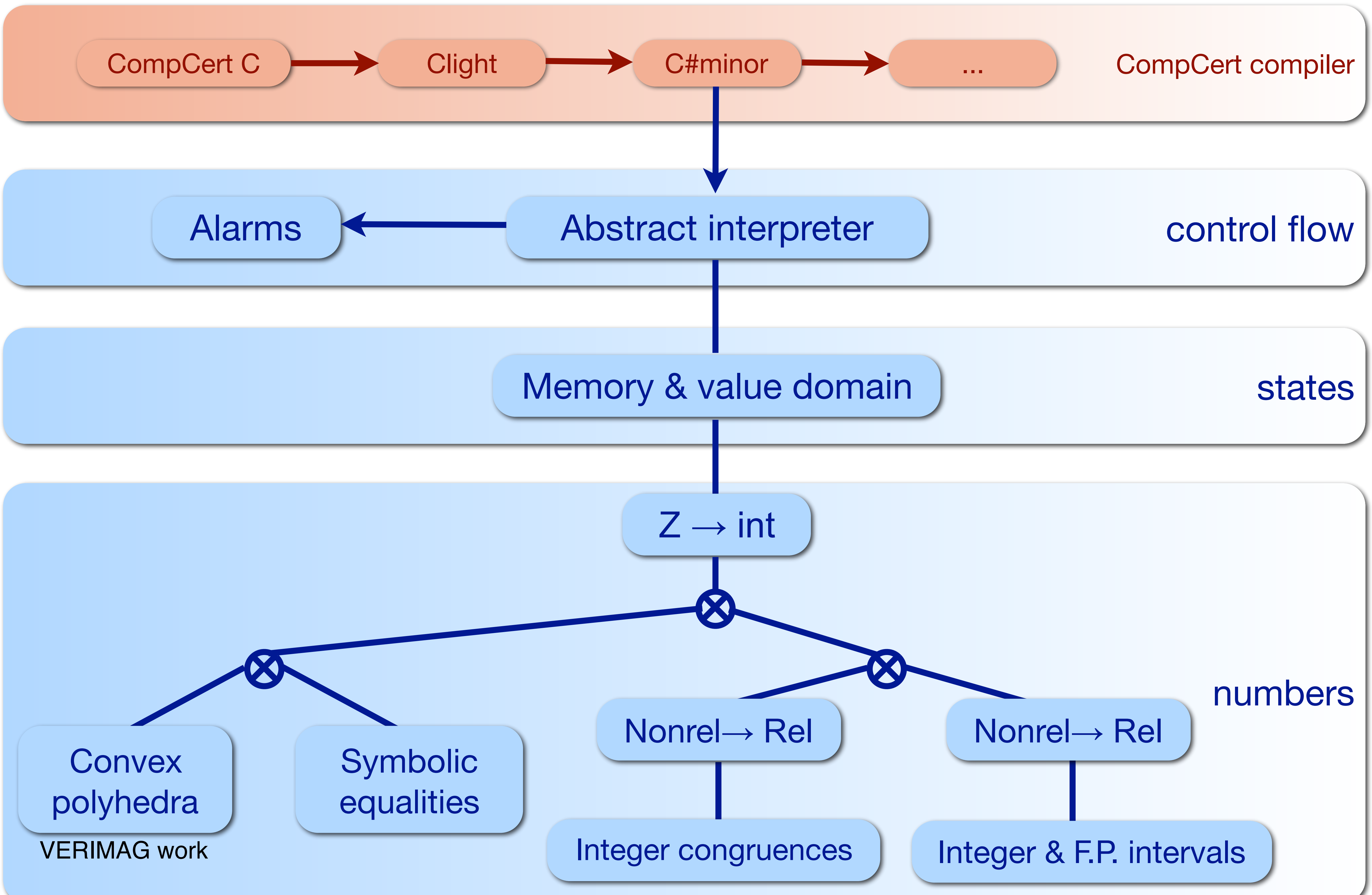
Verasco

Abstract numerical domains



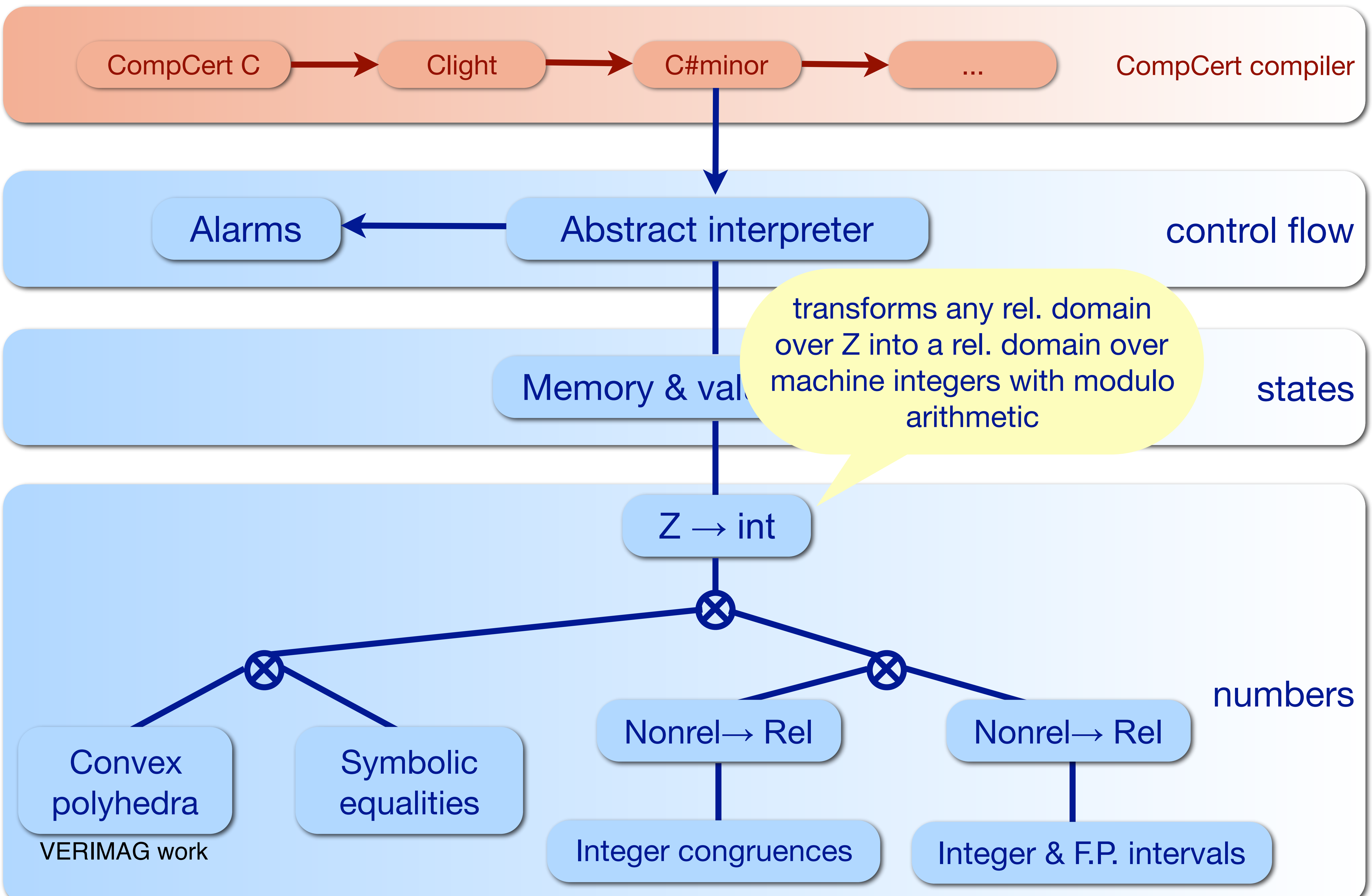
Verasco

Abstract numerical domains



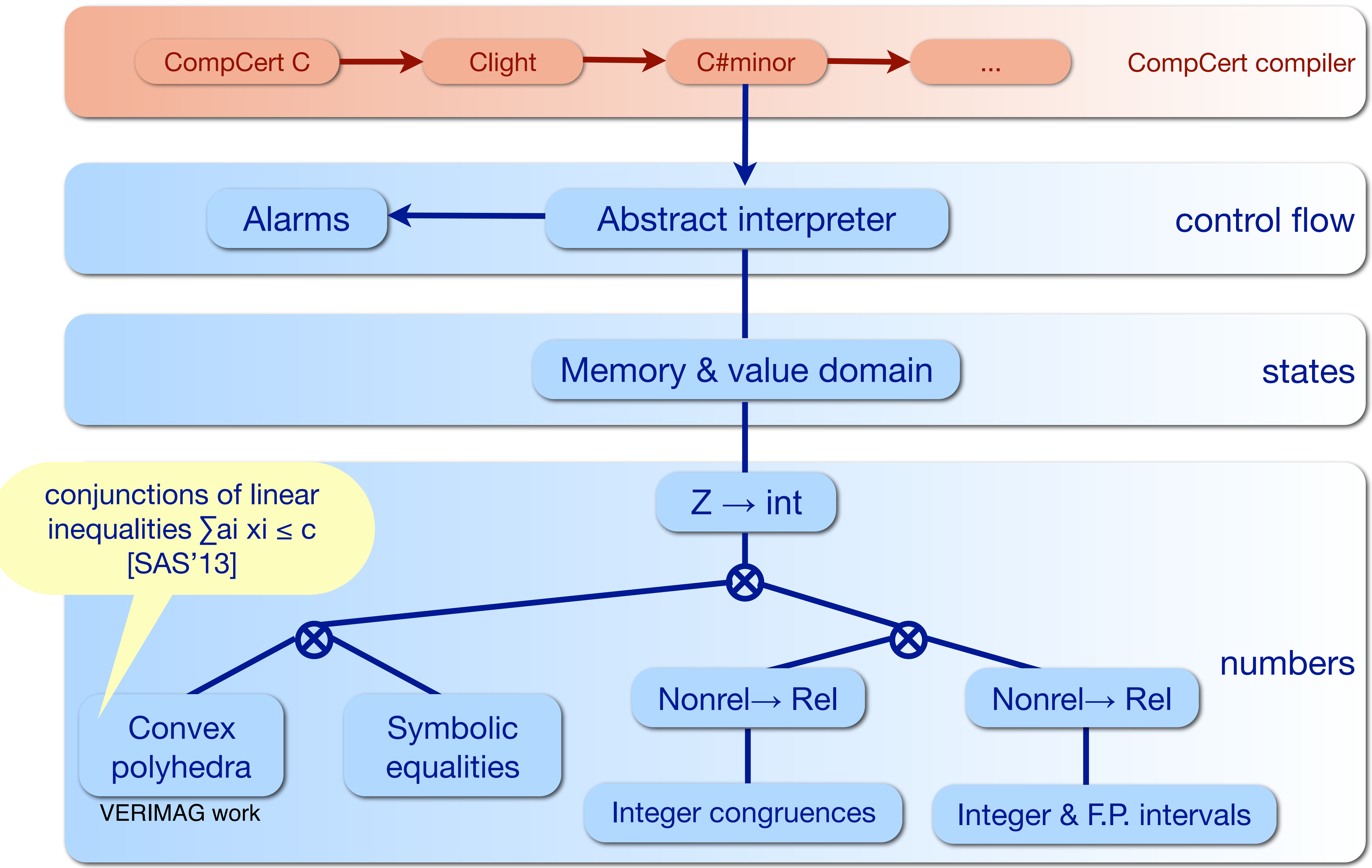
Verasco

Abstract numerical domains



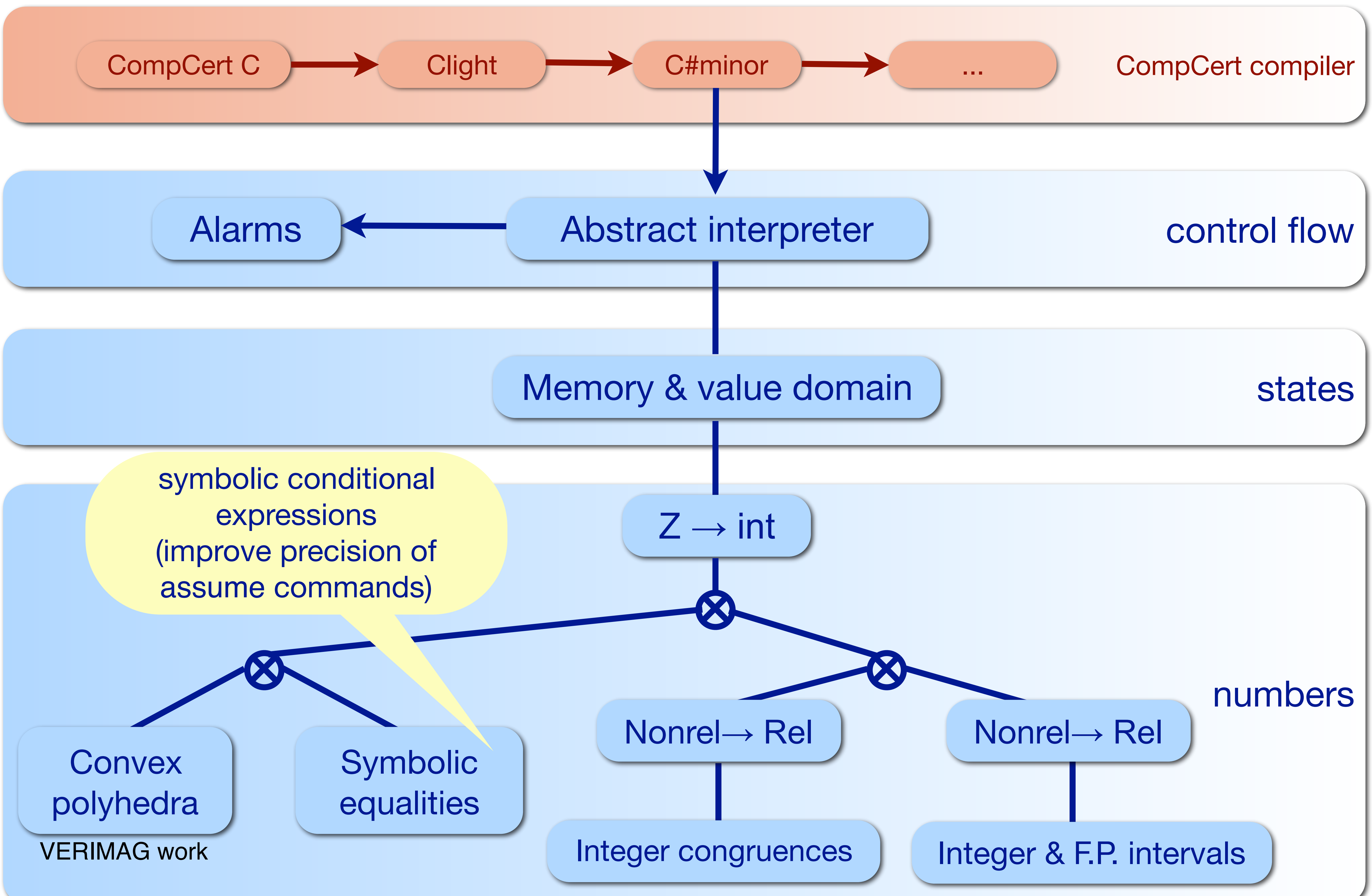
Verasco

Abstract numerical domains



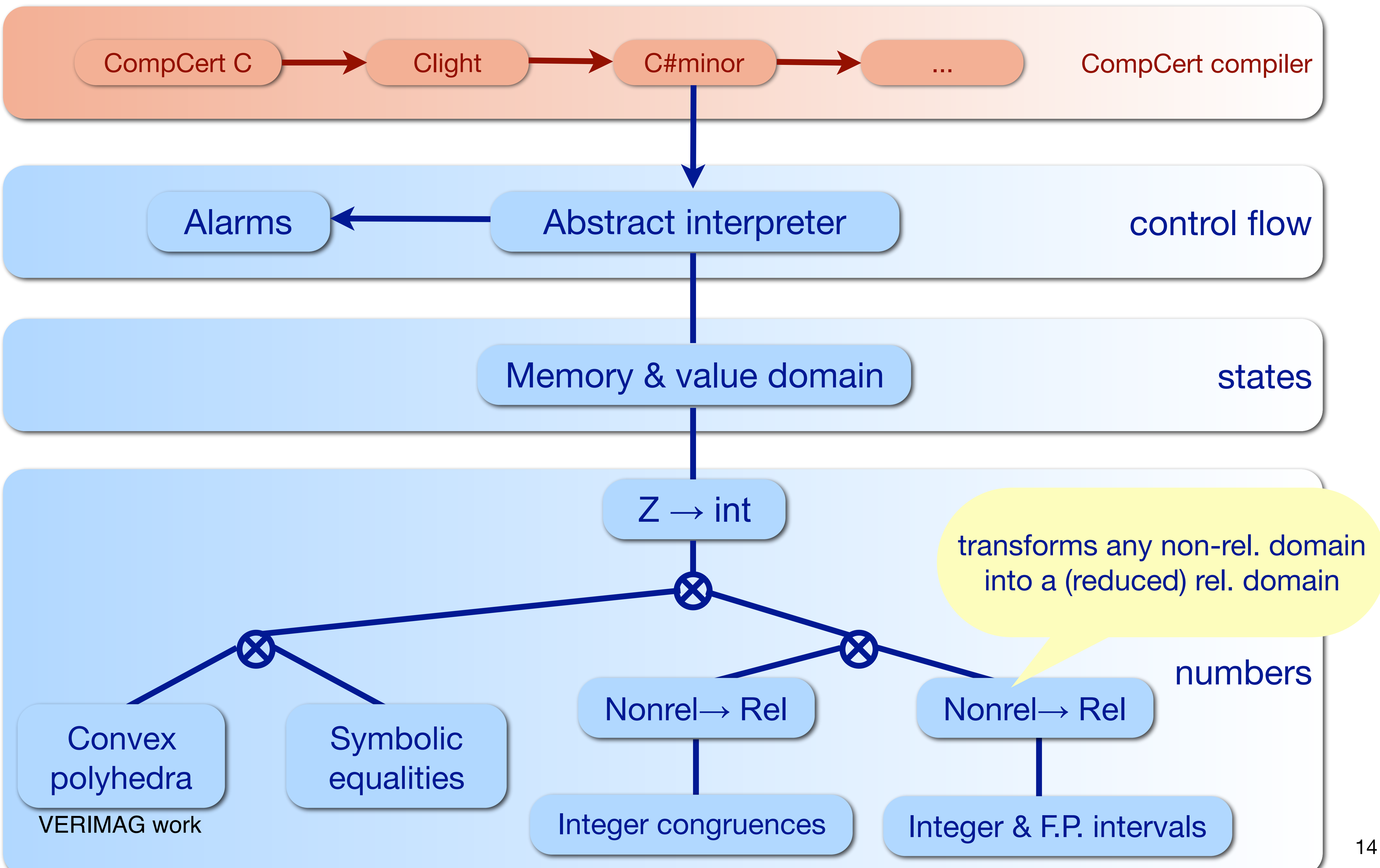
Verasco

Abstract numerical domains



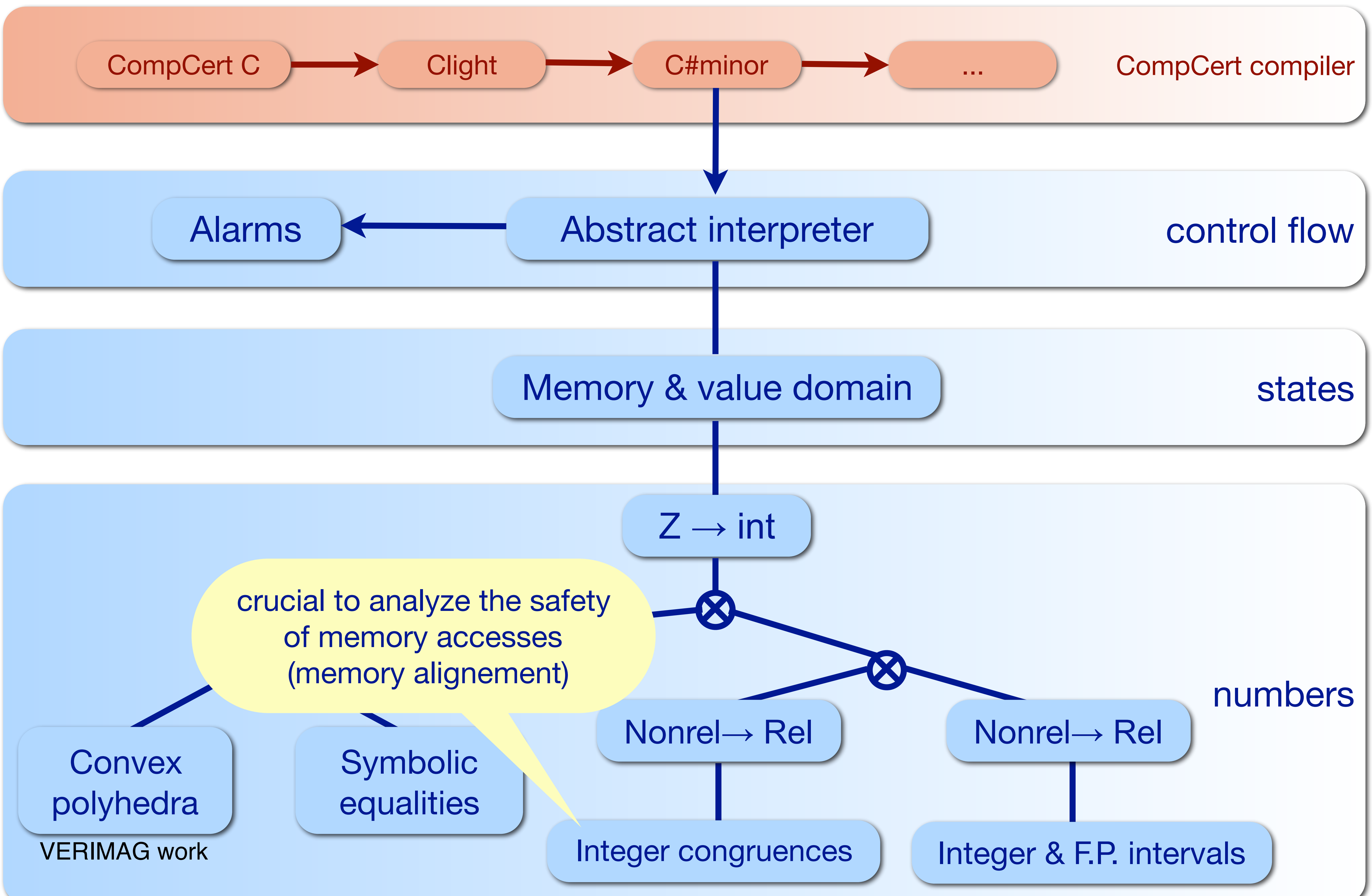
Verasco

Abstract numerical domains



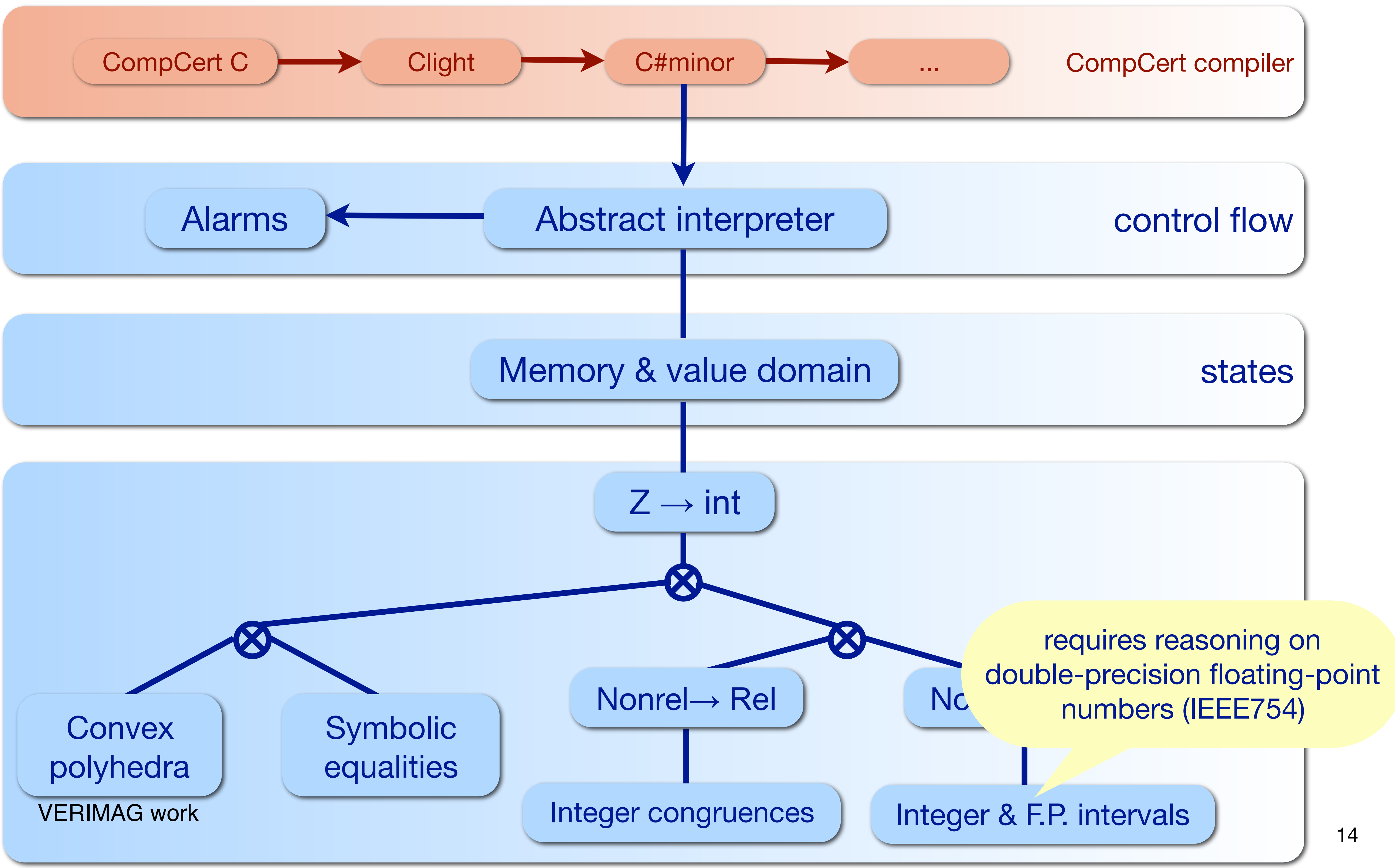
Verasco

Abstract numerical domains



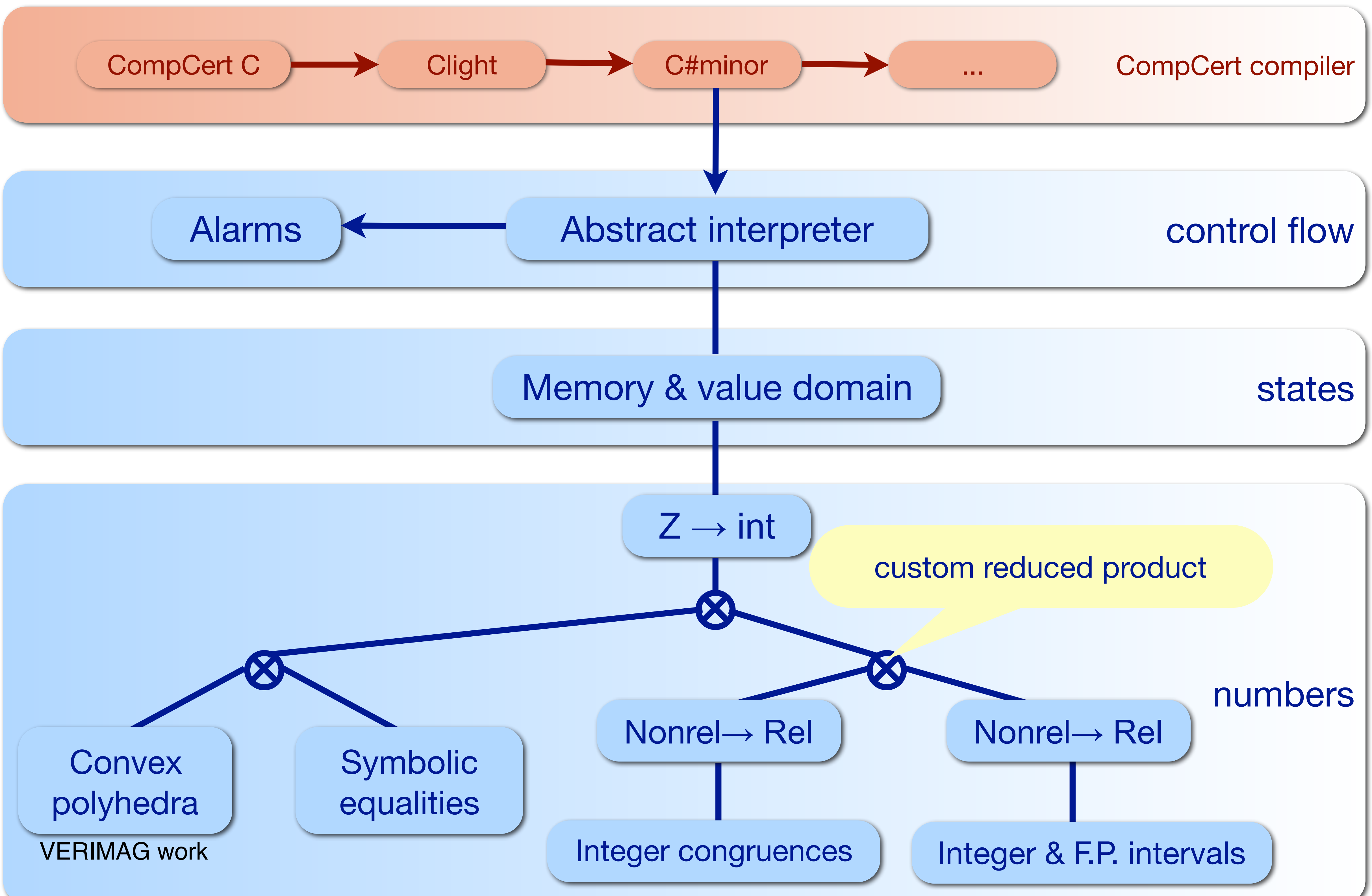
Verasco

Abstract numerical domains



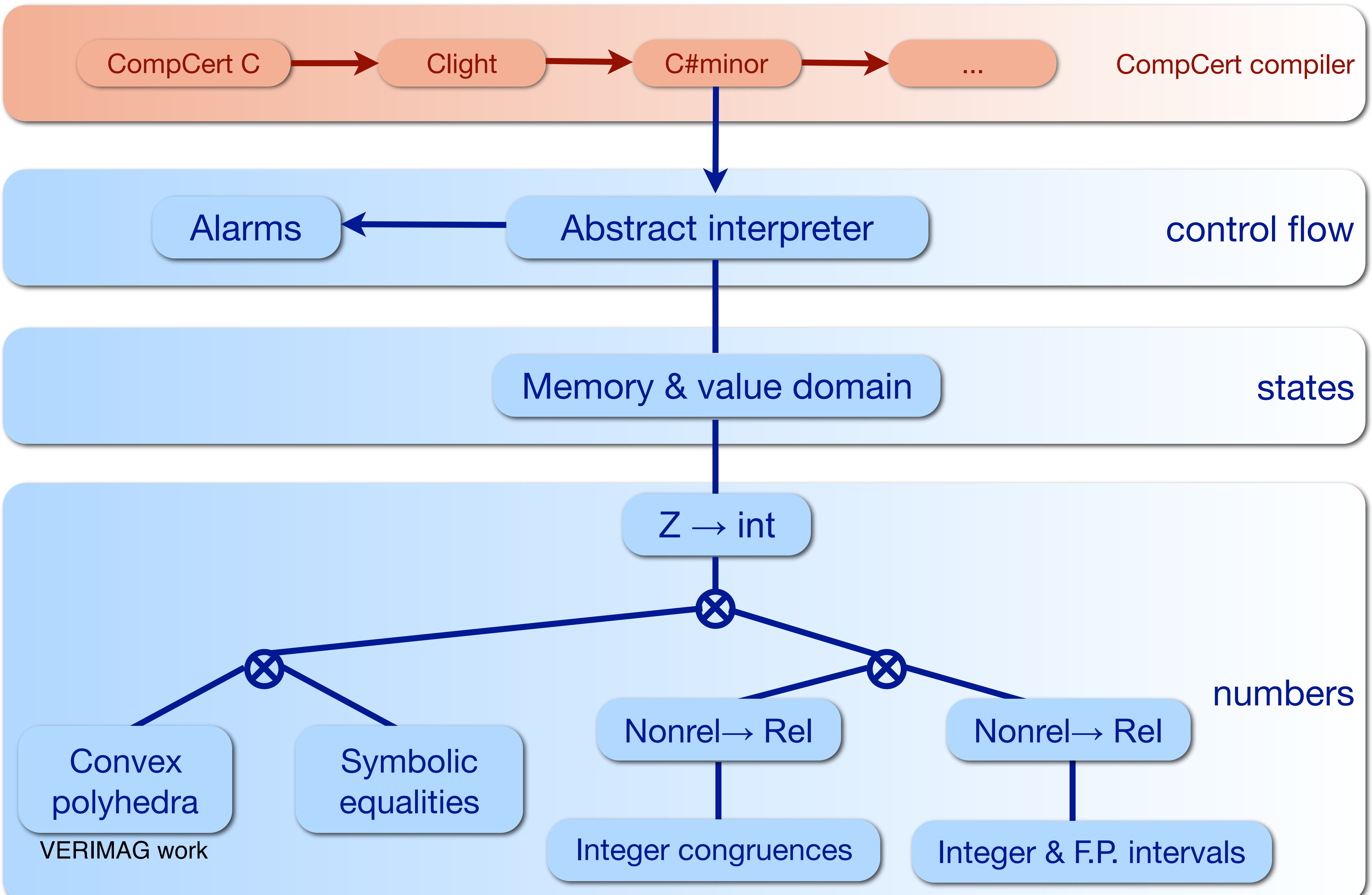
Verasco

Abstract numerical domains



Verasco

Abstract numerical domains



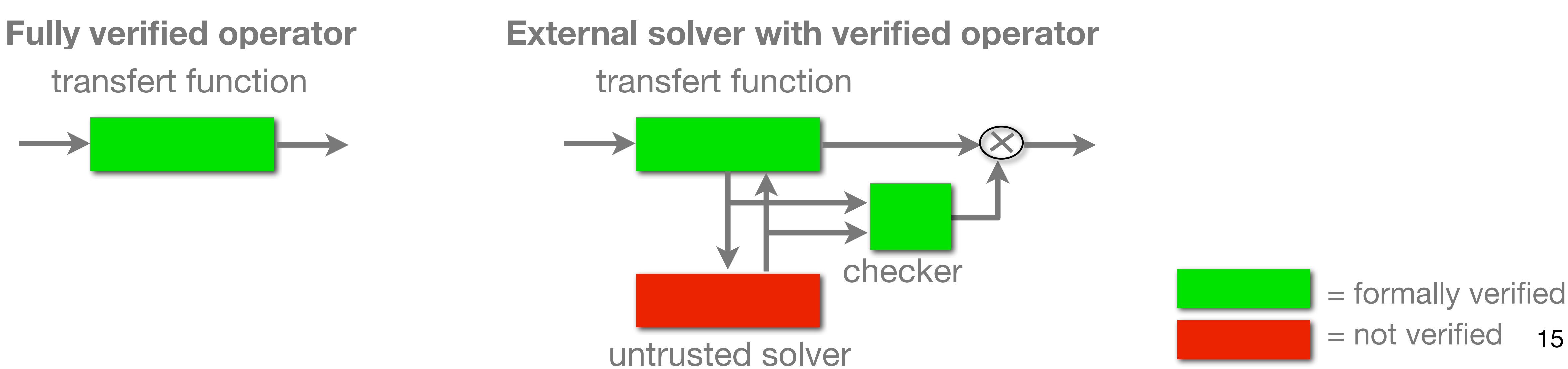
Verasco
Implementation

34 000 lines of Coq, excluding blanks and comments

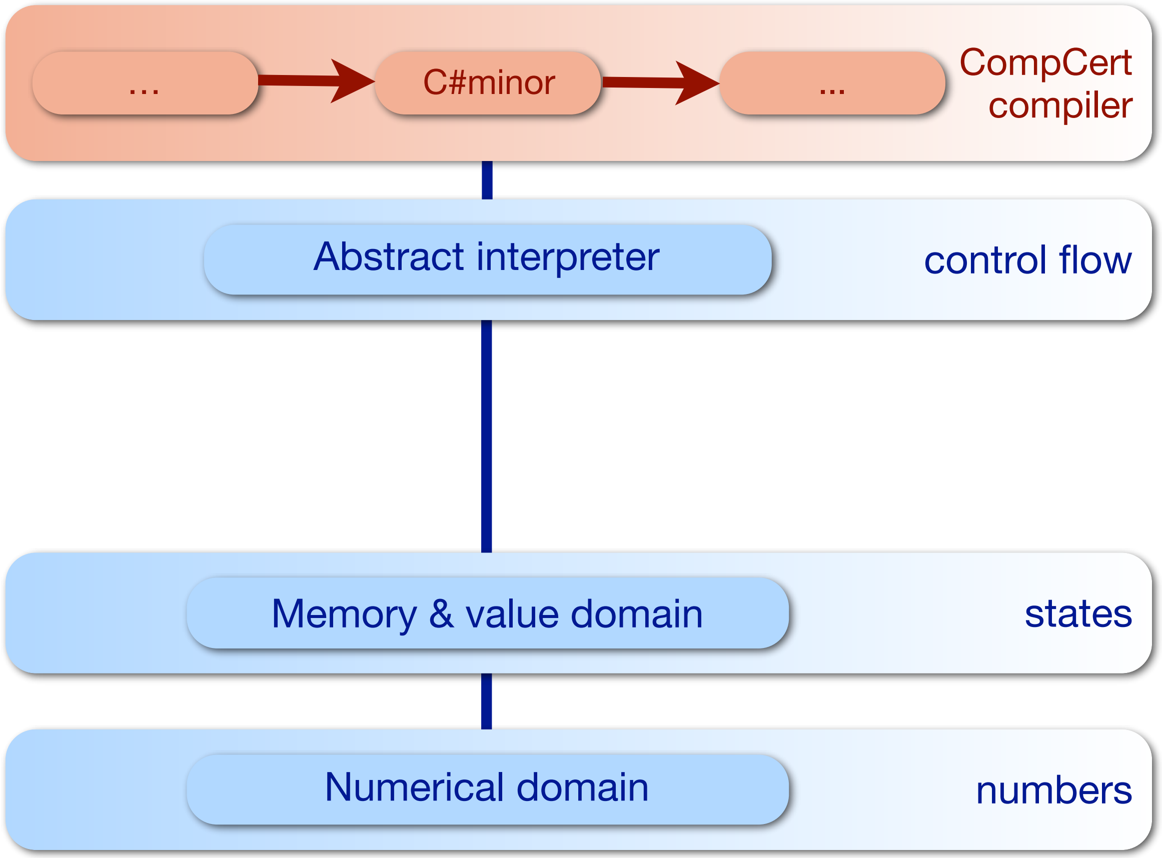
- half proof, half code & specs
- plus parts reused from CompCert

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.

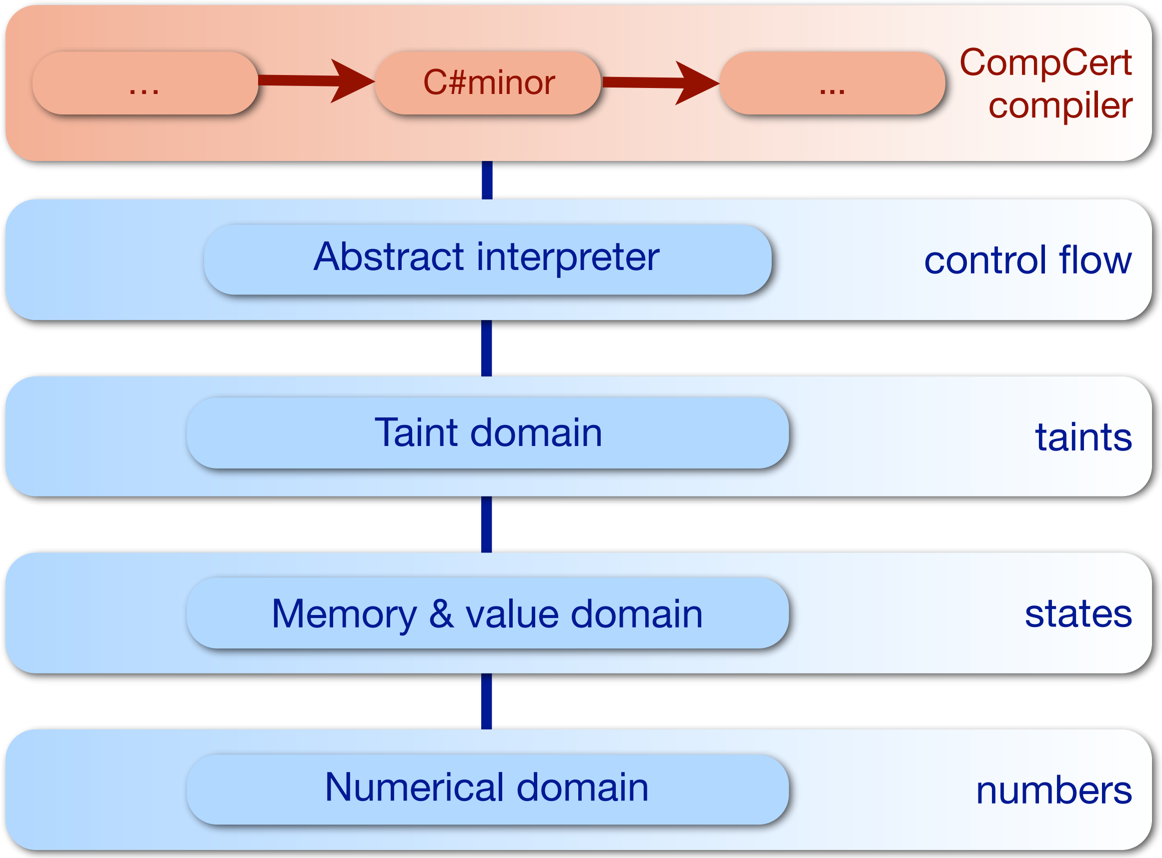


Constant-time analysis at source level



Constant-time analysis at source level

We design an *abstract functor*

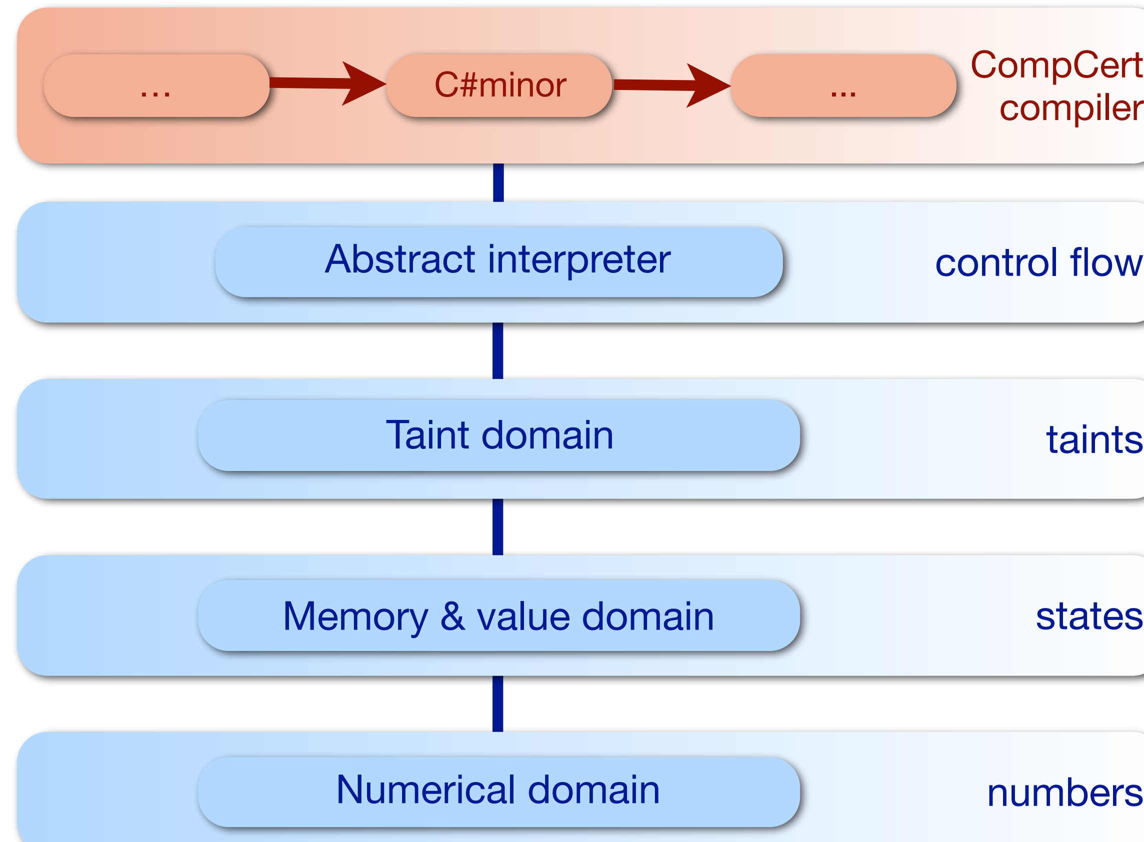


Constant-time analysis at source level

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*

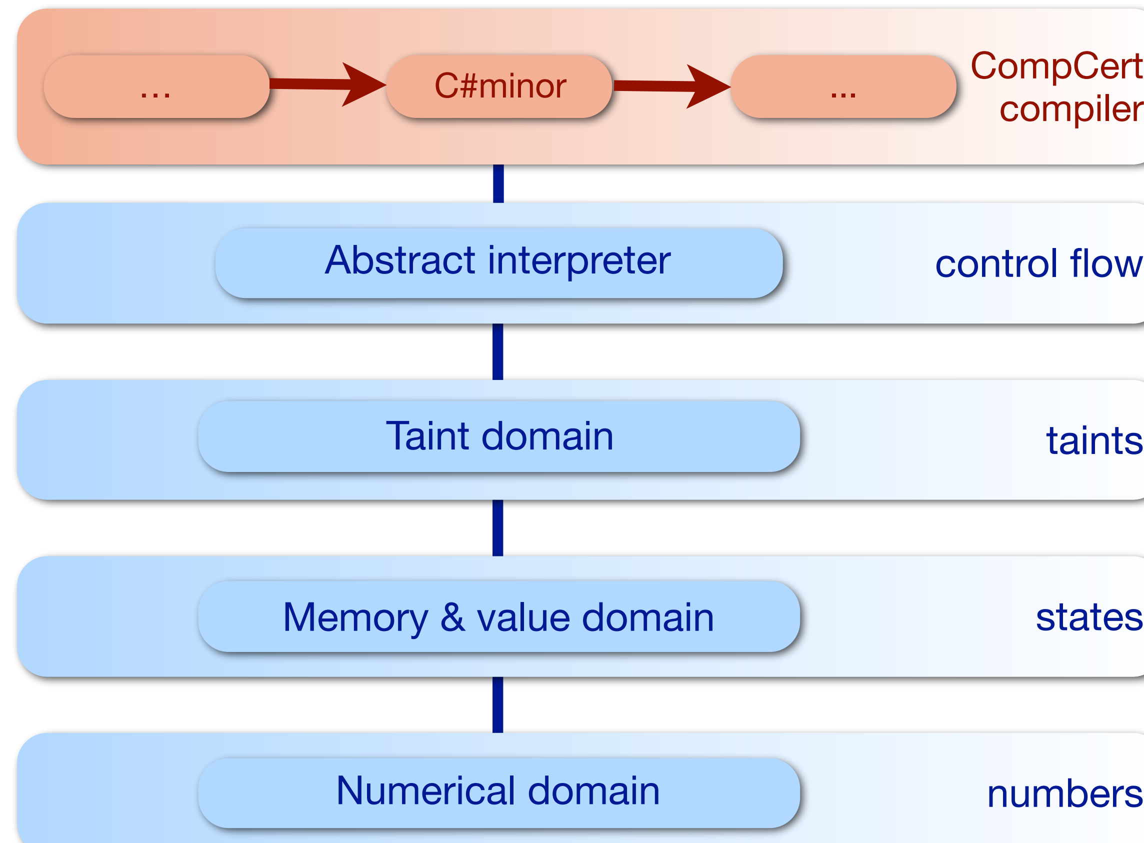
- takes as input an abstract memory domain

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

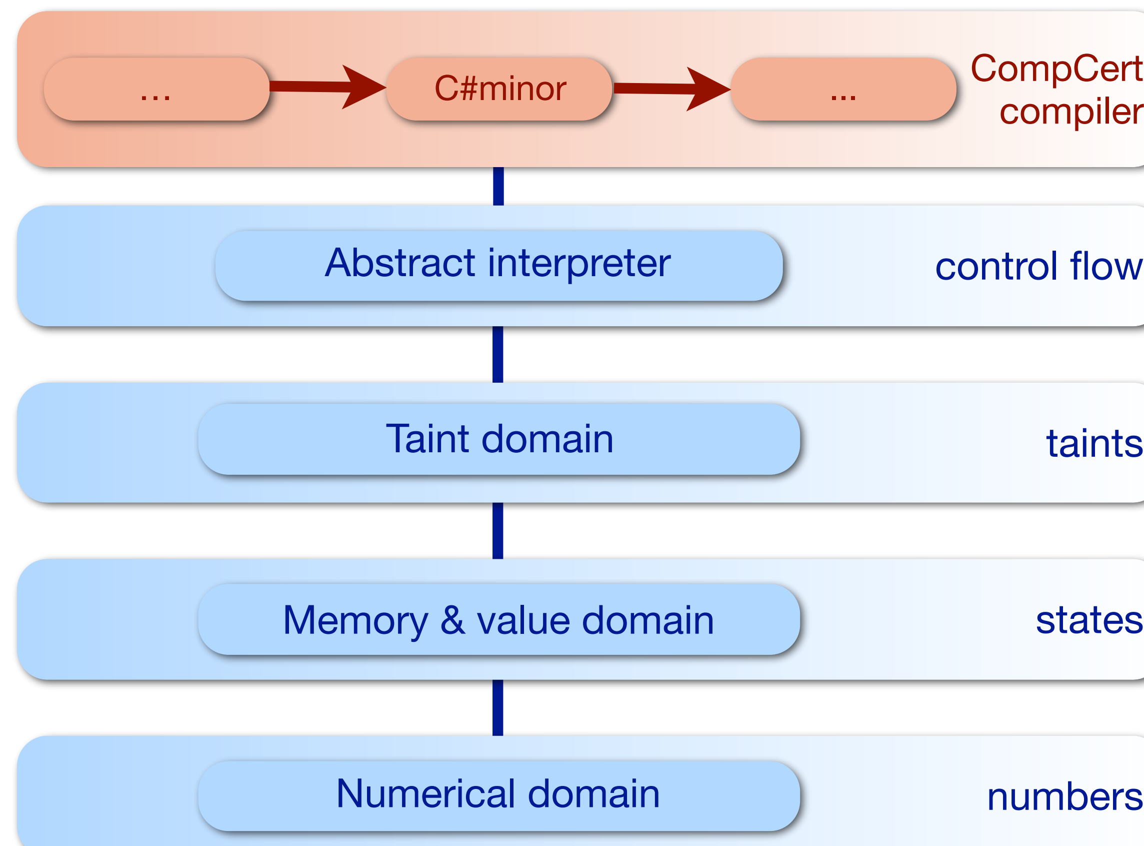
abstract value

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

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*

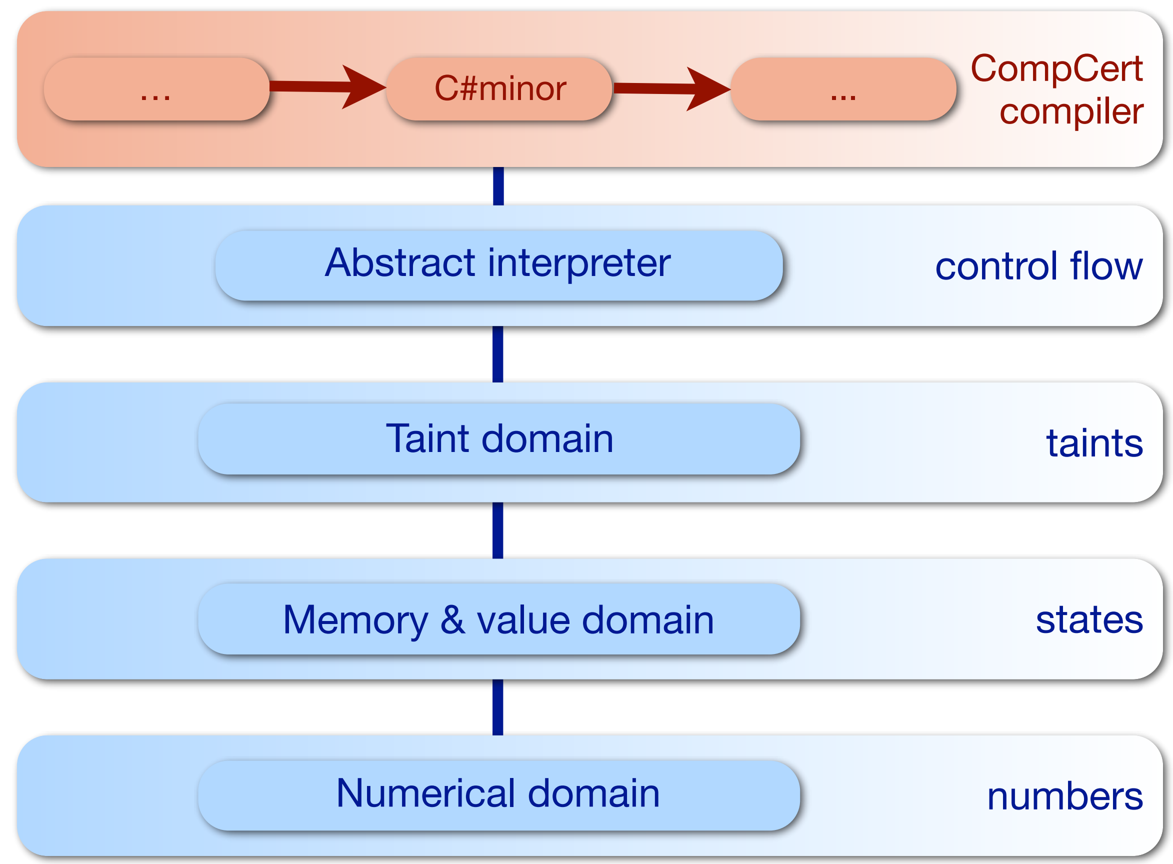
- takes as input an abstract memory domain

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

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

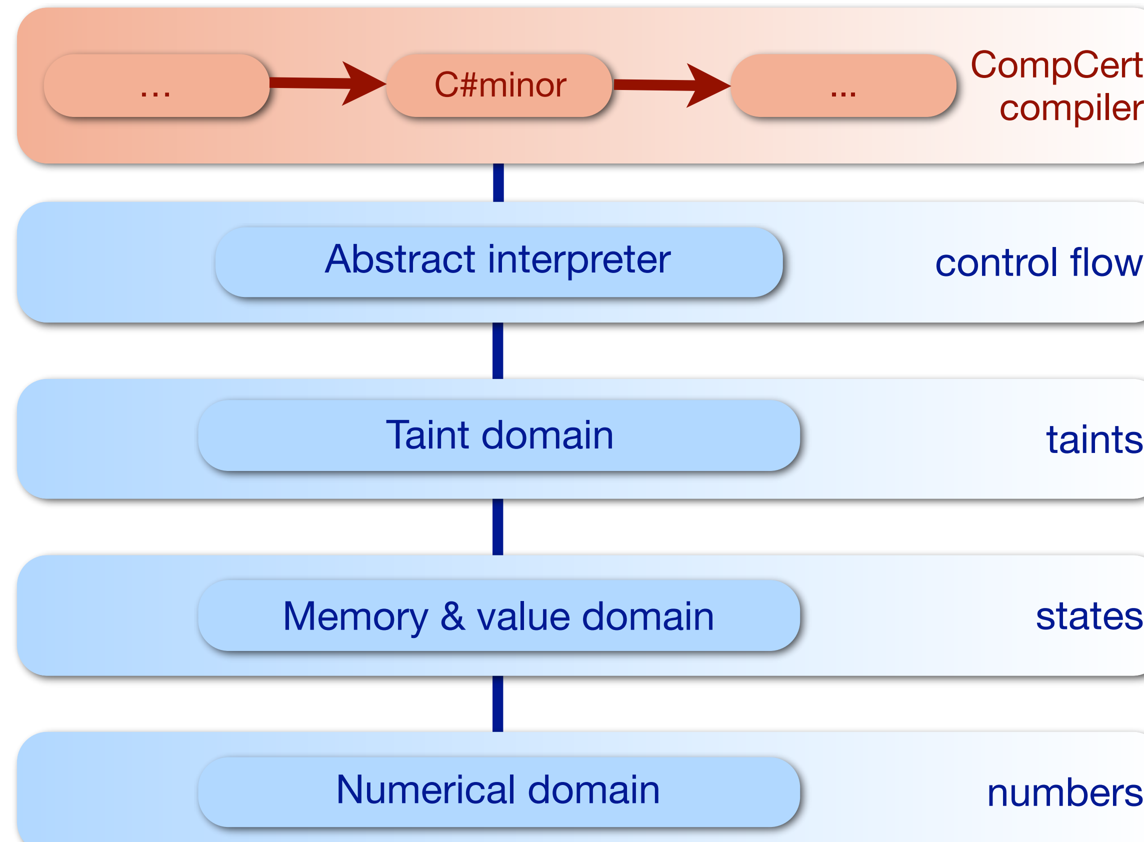
set of concrete memory locations

- 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.

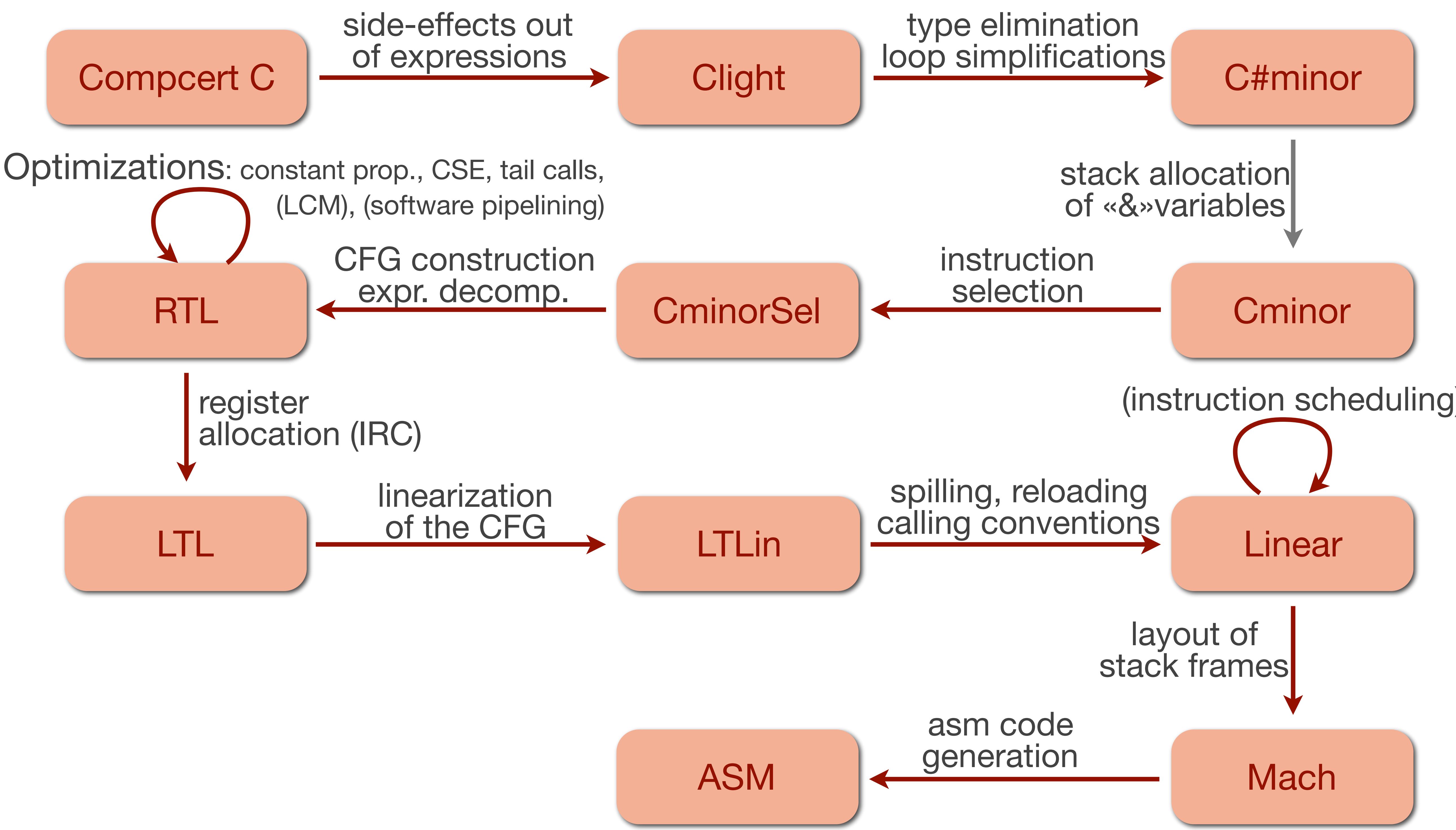
Not handled by Almeida et al. because LLVM alias analysis limitations

Preserving the property through compilation

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu.
Formal verification of a constant-time preserving C compiler.
POPL 2020.

- Makes precise what secure compilation means for cryptographic constant-time
- Provides a machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
- Explains how to turn a pre-exisiting formally-verified compiler into a formally-verified secure compiler
- Provides a proof toolkit for proving security preservation with simulation diagrams

CompCert: 1 compiler, 11 languages



CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

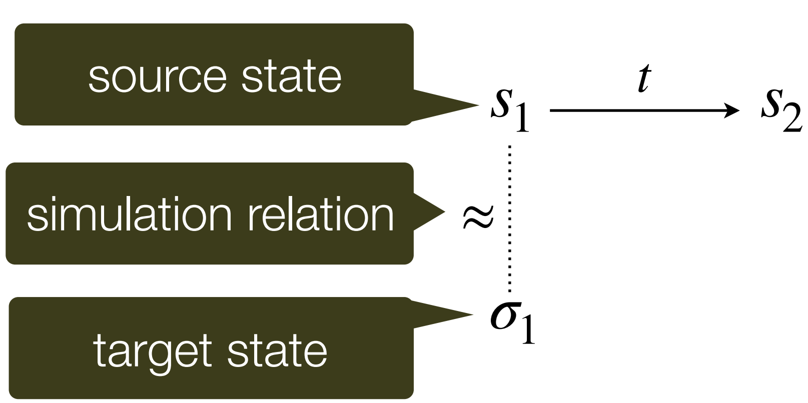
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

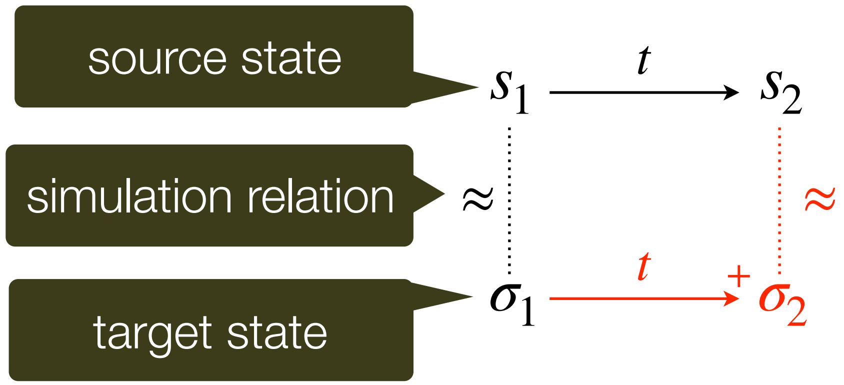


CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.

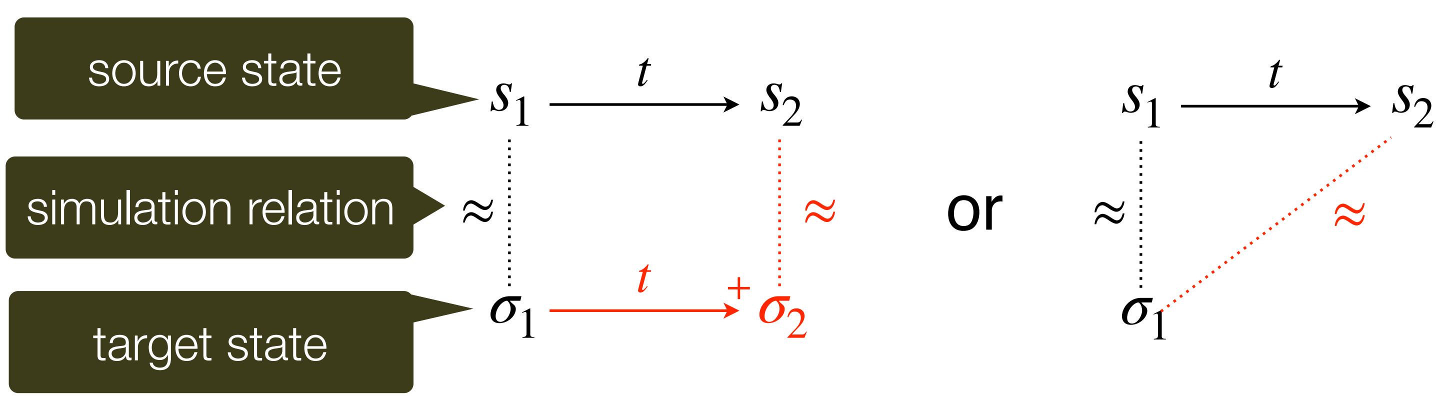
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



with $t = \varepsilon$

and $m(s_2) < m(s_1)$

well founded measure

CompCert:17 preservations proofs

Compiler pass	Explanation on the pass
Cshmgen	Type elaboration, simplification of control
Cminorgen	Stack allocation
Selection	Recognition of operators and addr. modes
RTLgen	Generation of CFG and 3-address code
Tailcall	Tailcall recognition
Inlining	Function inlining
Renumber	Renumbering CFG nodes
ConstProp	Constant propagation
CSE	Common subexpression elimination
Deadcode	Redundancy elimination
Allocation	Register allocation
Tunneling	Branch tunneling
Linearize	Linearization of CFG
CleanupLabels	Removal of unreferenced labels
Debugvar	Synthesis of debugging information
Stacking	Laying out stack frames
Asmgen	Emission of assembly code

Cryptographic constant-time property: defining leakages

- We enrich the CompCert traces of events with **leakages** of two types
  - either the truth value of a condition,
  - or a pointer representing the address of
    - either a memory access (i.e., a load or a store)
    - or a called function
- Using **event erasure**, from $s \xrightarrow{t} s'$ we can extract
  - the compile-only judgment $s \xrightarrow{t}_{\text{comp}} s'$
  - the leak-only judgment $s \xrightarrow{t}_{\text{leak}} s'$
- **Program leakage** is defined as the behavior of the $\rightarrow_{\text{leak}}$ semantics

Cryptographic constant-time property: preservation

- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

$$\begin{tikzcd} s \arrow[r, "t"] \arrow[d, dash, "\varphi"'] & {}^{*}\text{leak} \\ s' \arrow[r, "t'"] & {}^{*}\text{leak} \end{tikzcd} \quad \text{implies} \quad t = t'$$

Cryptographic constant-time property: preservation

- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

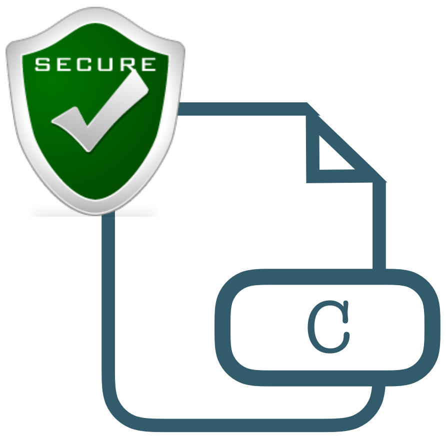
Main Theorem (Constant-Time security preservation): Let P be a safe Clight source program that is compiled into an x86 assembly program P'. If P is constant-time w.r.t. φ, then so is P'.

Conclusion

Conclusion

this talk focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like)

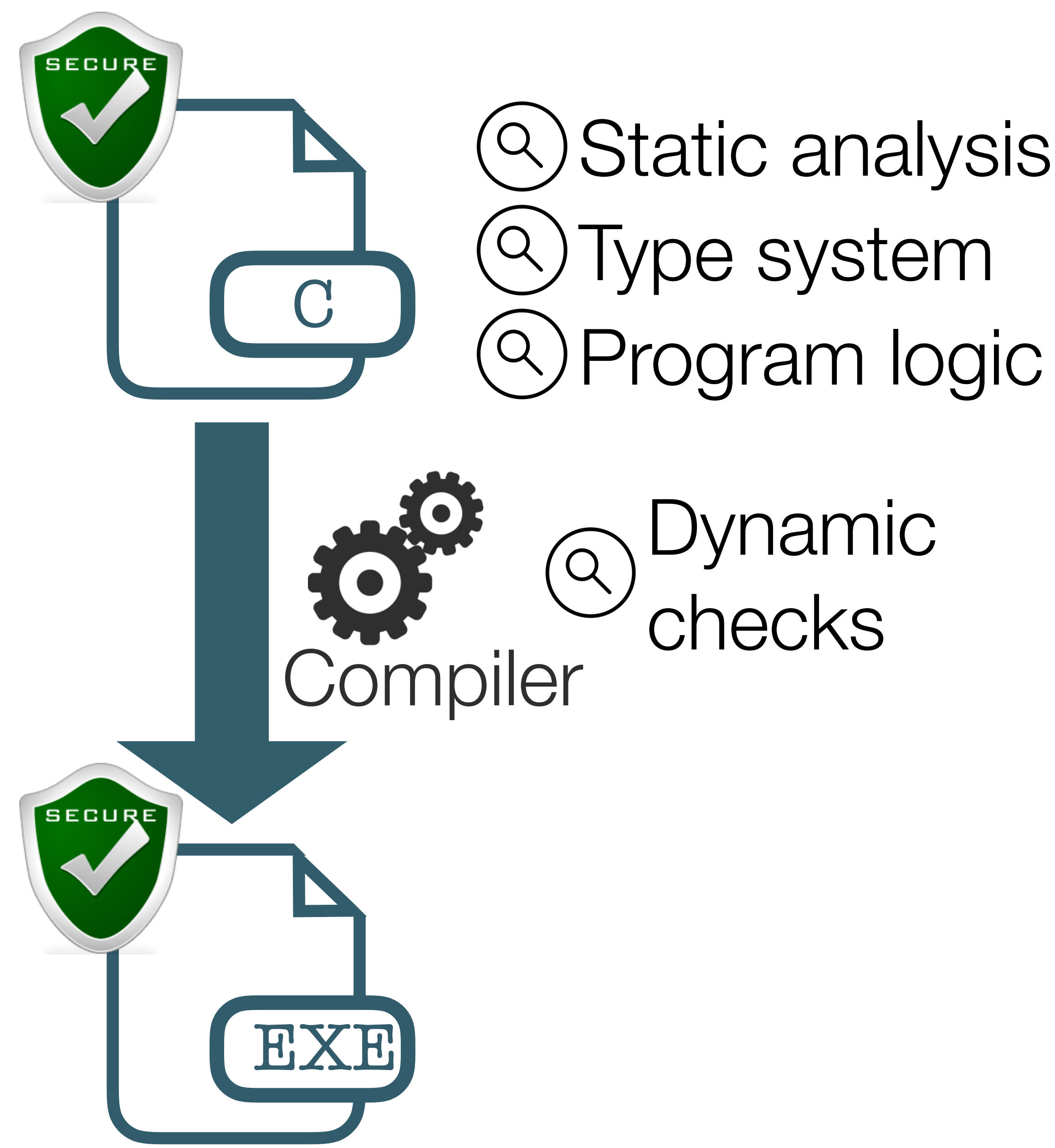


- Static analysis
- Type system
- Program logic

Conclusion

this talk focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like)
- We make sure the compiler will generate executables that are as secure



Conclusion

this talk focused on Crypto-Constant-Time security property

- We can build secure programming abstractions at source level (C-like)
- We make sure the compiler will generate executables that are as secure
- We reduce as much as possible the TCB (Trusted Computing Base) with formal proofs