Building Verified Program Analyzers in Coq

Lecture 4: A verified value analysis for CompCert

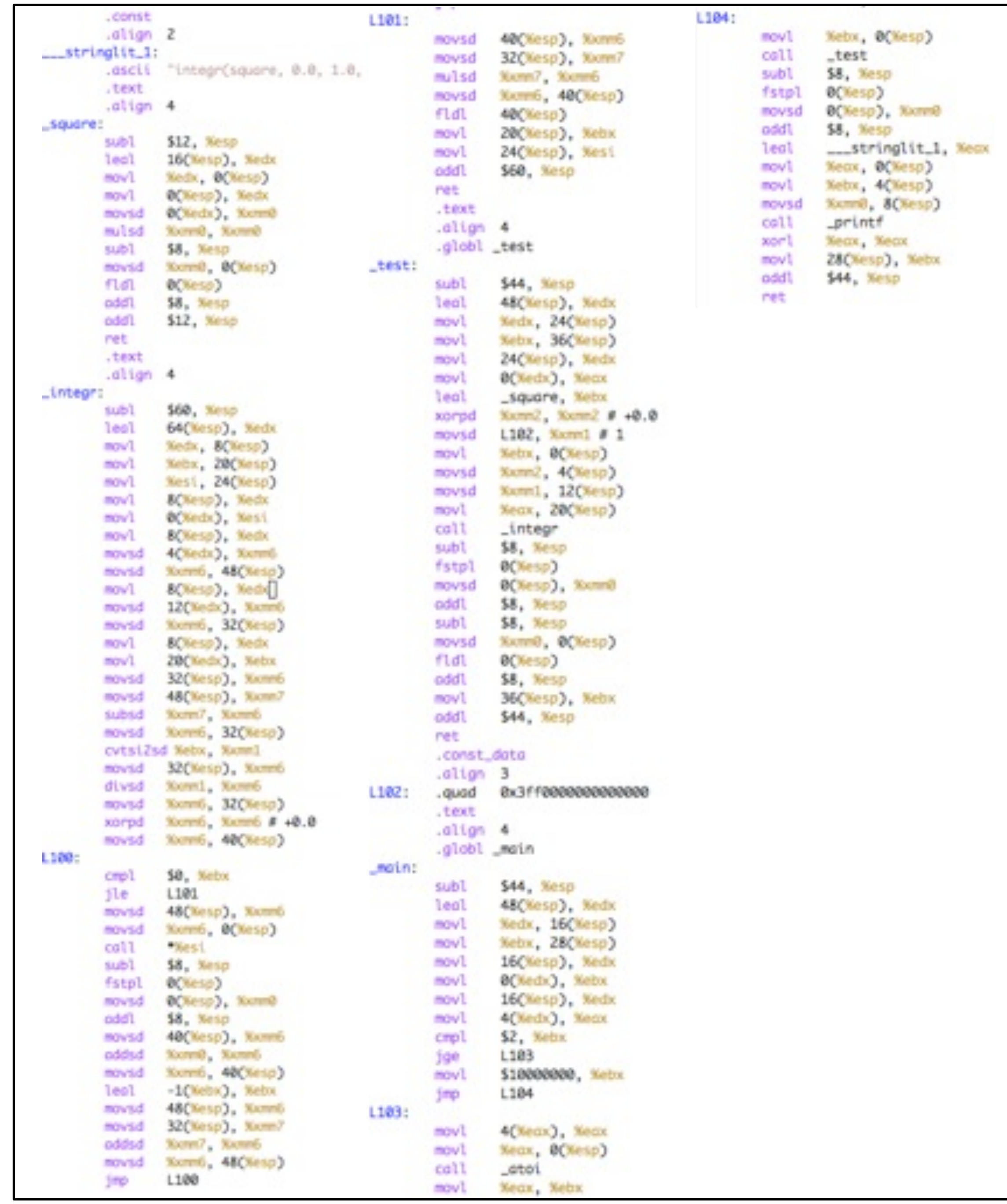
David Pichardie - INRIA Rennes / Harvard University

The CompCert verified C compiler

Compiler built and proved by Xavier Leroy et al.

Slides largely inspired by Leroy's own material

Critical embedded software

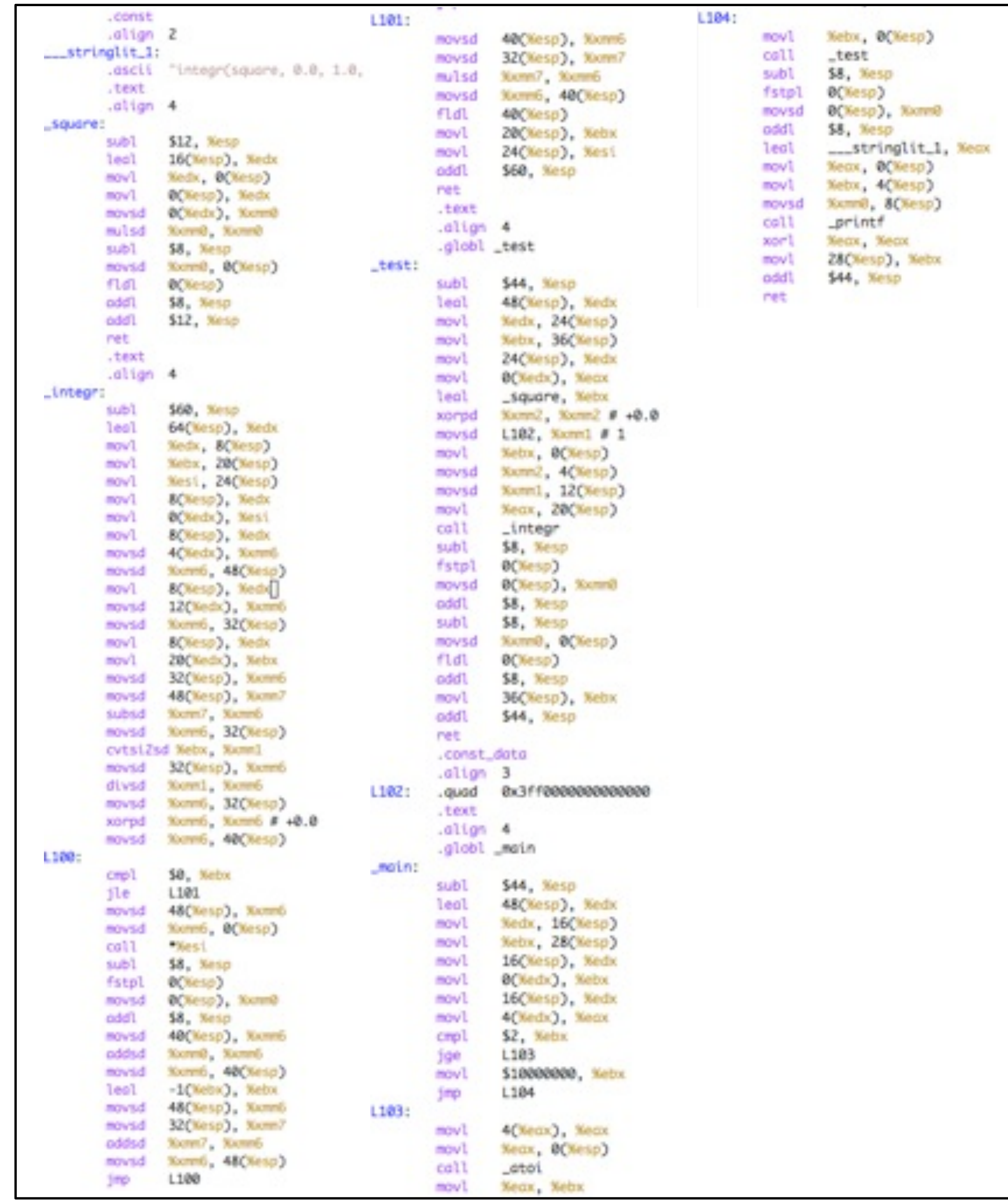


critical_prog.ppc

Critical embedded software

High degree of assurance is required

- is the program critical_prog.ppc safe ?



critical_prog.ppc

Critical embedded software

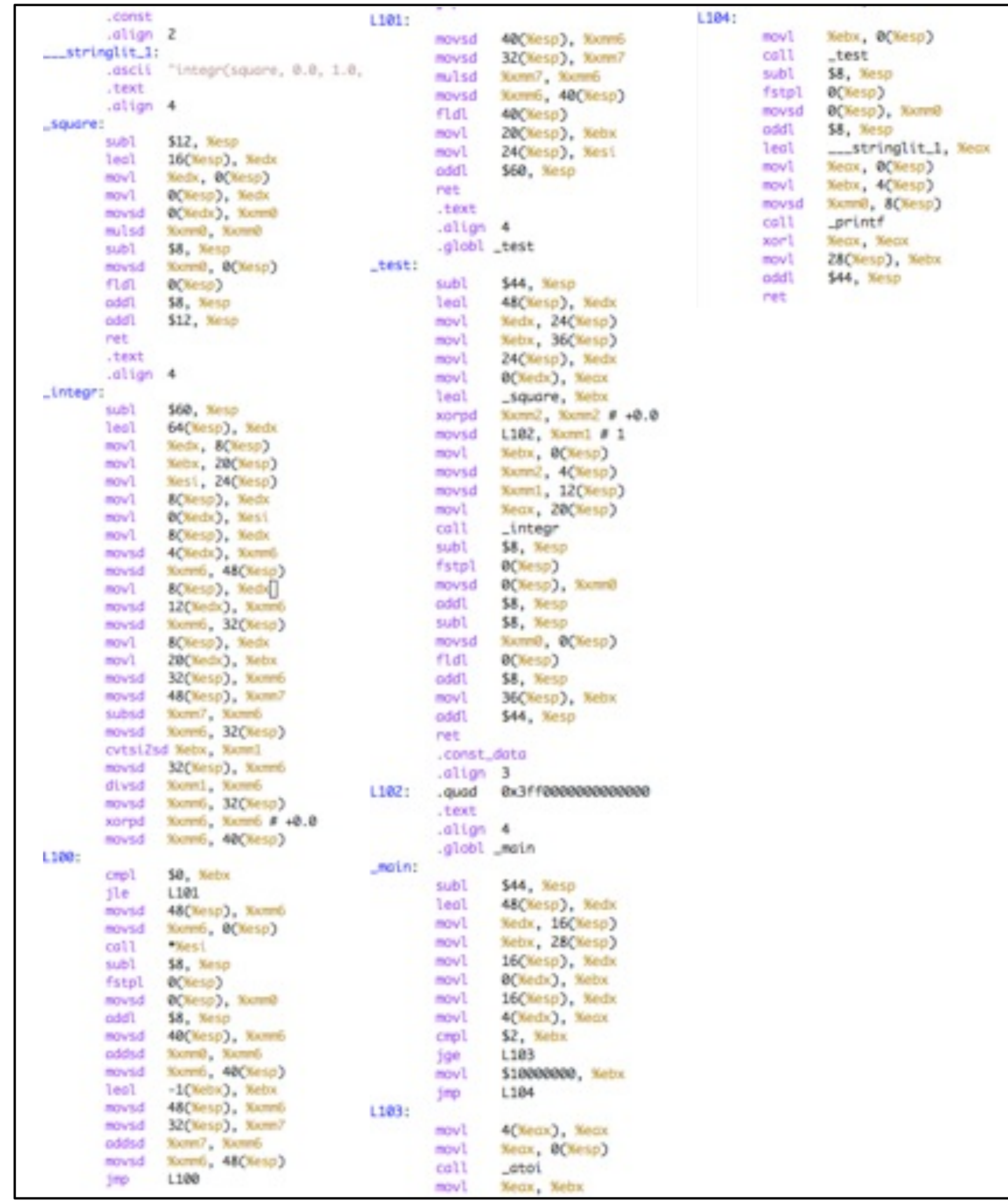
High degree of assurance is required

- is the program critical_prog.ppc safe ?

2 options:

- qualify the PPC program as if hand-written (intensive testing, painful manual review...)

- qualify the program at the source level (static analysis, model checking, or program proof)



`critical_prog.ppc`

Critical embedded software

High degree of assurance is required

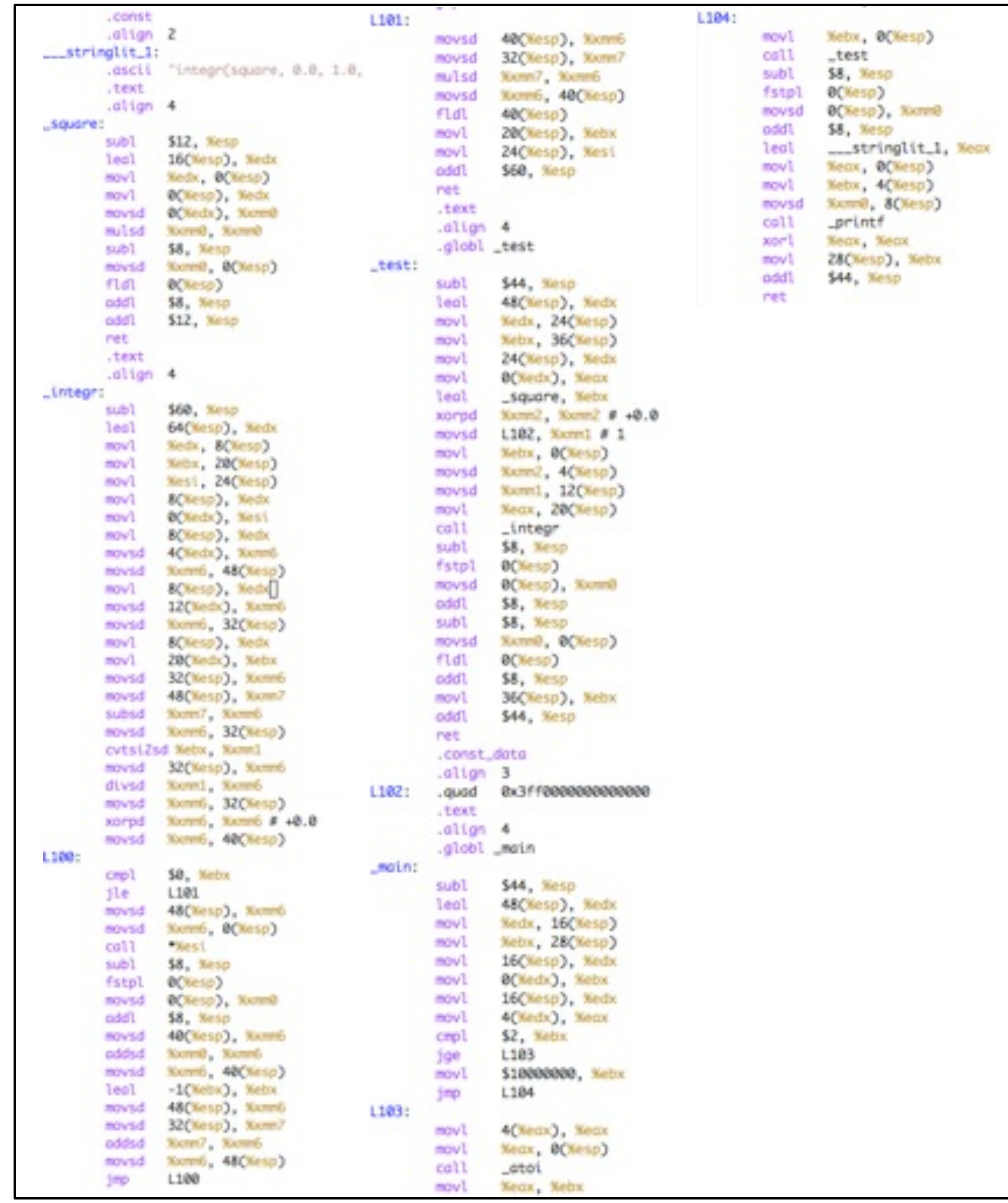
- is the program critical_prog.ppc safe ?

2 options:

- qualify the PPC program as if hand-written (intensive testing, painful manual review...)
- qualify the program at the source level (static analysis, model checking, or program proof)

2nd option is preferred in practice

- can you trust your compiler ?
- this talk: apply formal verification techniques to the compiler itself !



`critical_prog.ppc`

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

> *We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]*

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

> *We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]*
>
> *We found and reported hundreds of previously unknown bugs. [...]*

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

Most of the bugs we found in GCC were in the middle end: the machine- independent optimizers.

Miscompilation happens

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

> *We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]*
>
> *We found and reported hundreds of previously unknown bugs. [...]*
>
> *Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.*
>
> *Most of the bugs we found in GCC were in the middle end: the machine- independent optimizers.*

Optimizing compilers rely on complex static analyses!

The CompCert project

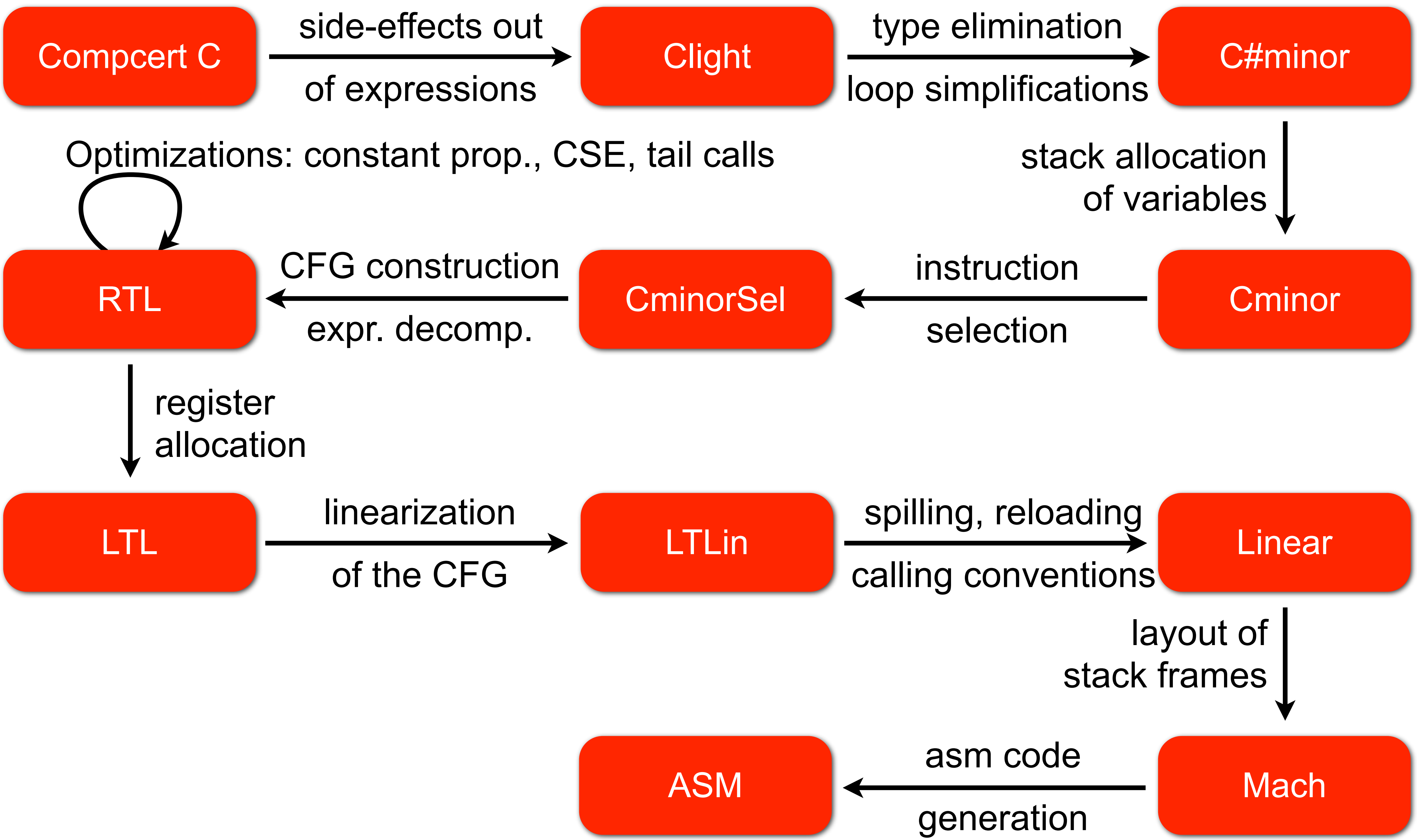
X. Leroy, S. Blazy et al.

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  ⇒ careful code generation; some optimisations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler
(otherwise see Zdancewic et al’s Verified LLVM project).

The formally verified part of the compiler



Formally verified in Coq

After 50 000 lines of Coq and 4 person.years of effort

```
Theorem transf_c_program_is_refinement:
  ∀ p tp,
  transf_c_program p = OK tp →
  (∀ beh, exec_C_program p beh → not_wrong beh) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

Behaviors `beh` = termination / divergence / going wrong
+ trace of I/O operations (syscalls, volatile accesses).

Compiler verification patterns
(for each pass)

Compiler verification patterns
(for each pass)

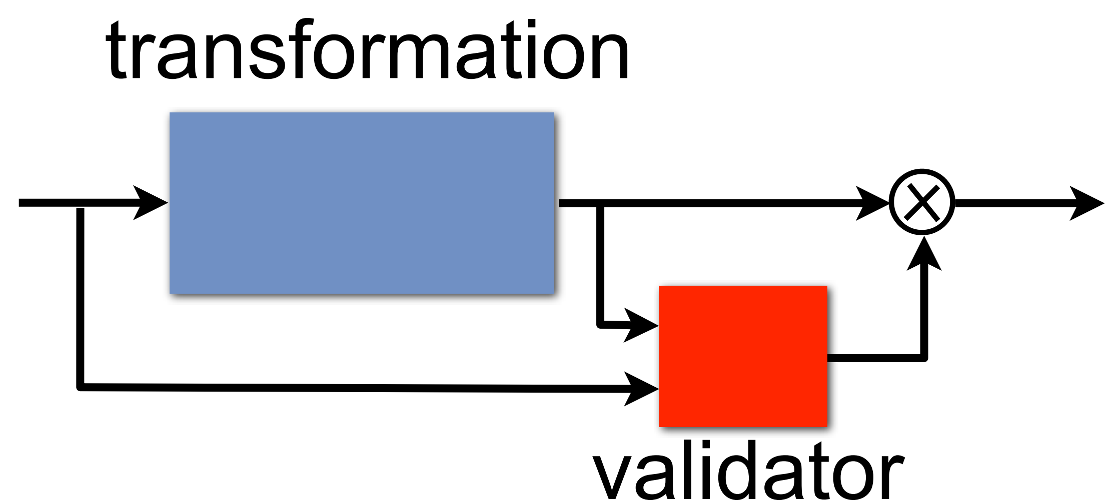
Verified transformation

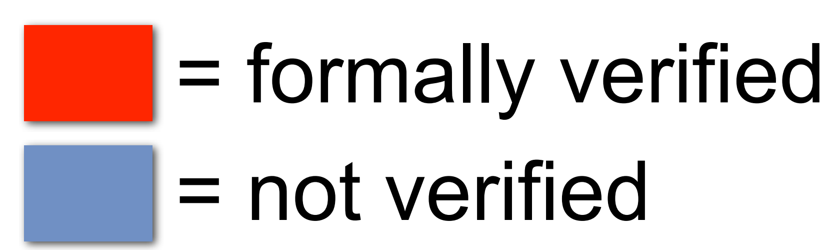
Compiler verification patterns

(for each pass)

Verified transformation

Verified translation validation



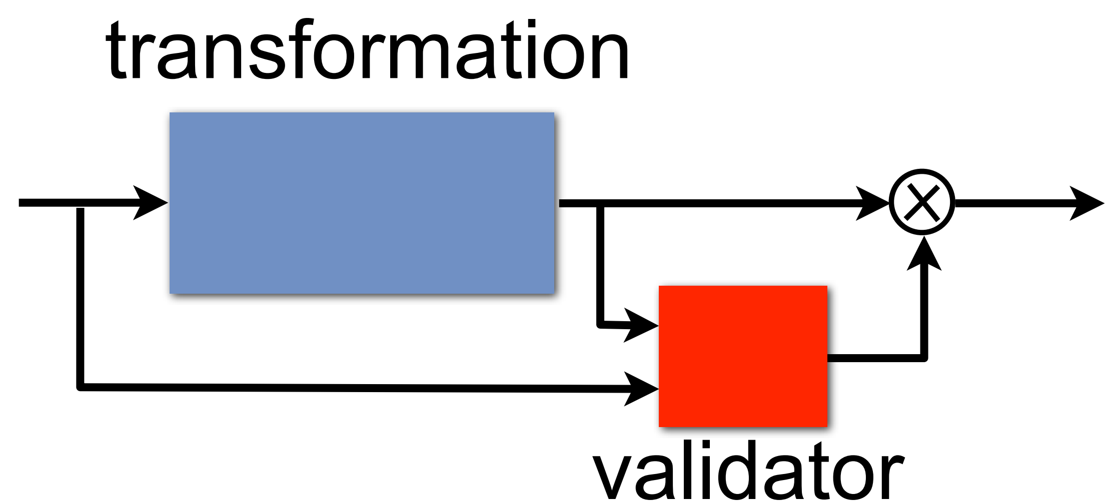


Compiler verification patterns

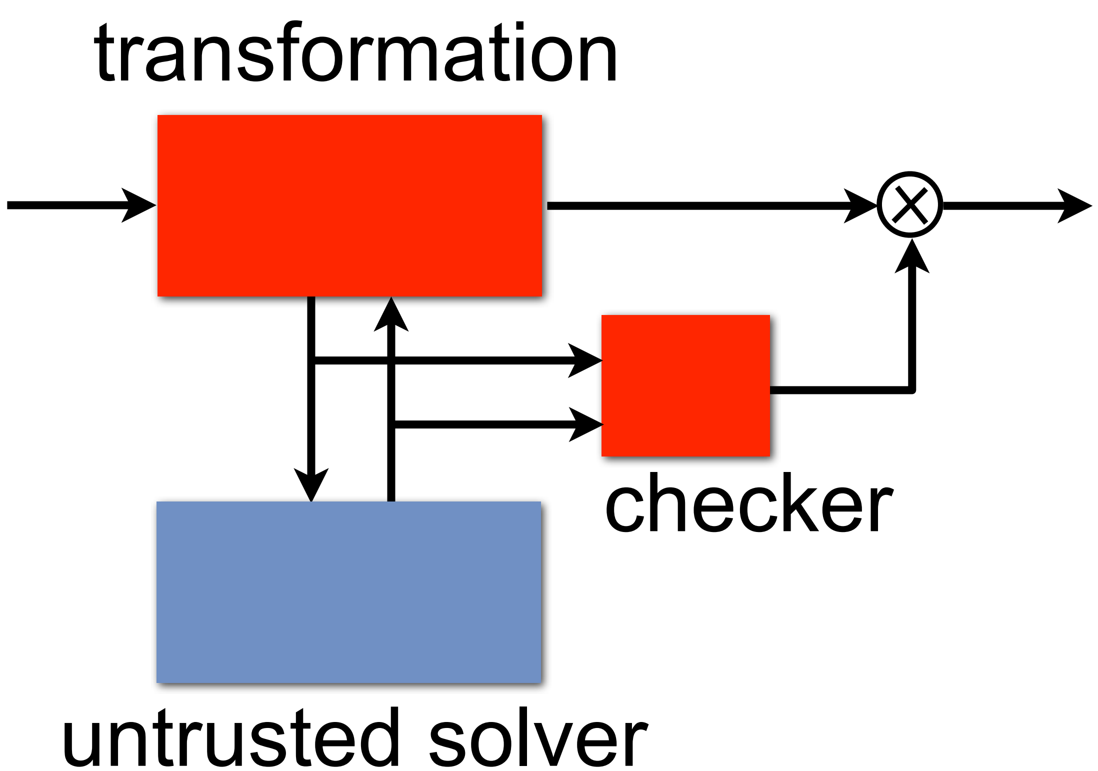
(for each pass)

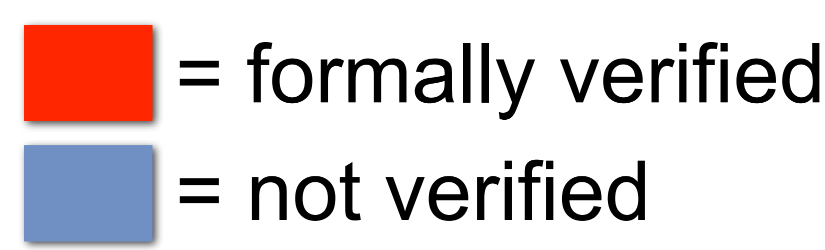
Verified transformation

Verified translation validation



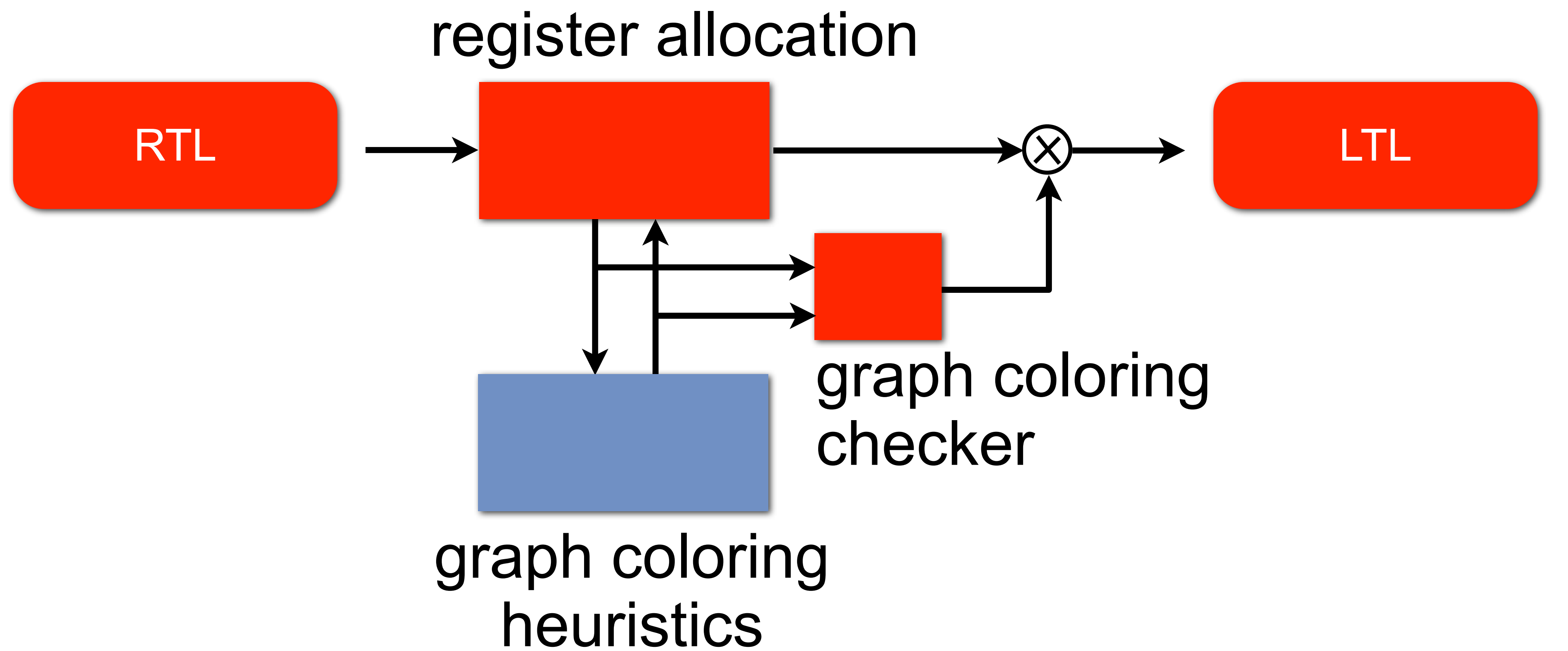
External solver with verified validation





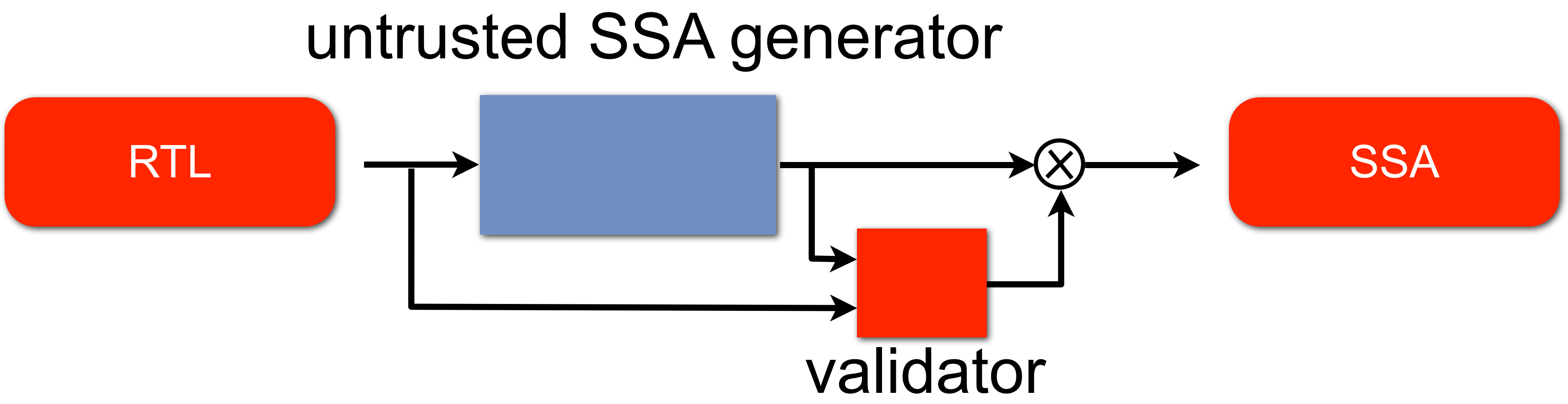
External solver with verified validation

Example: register allocation



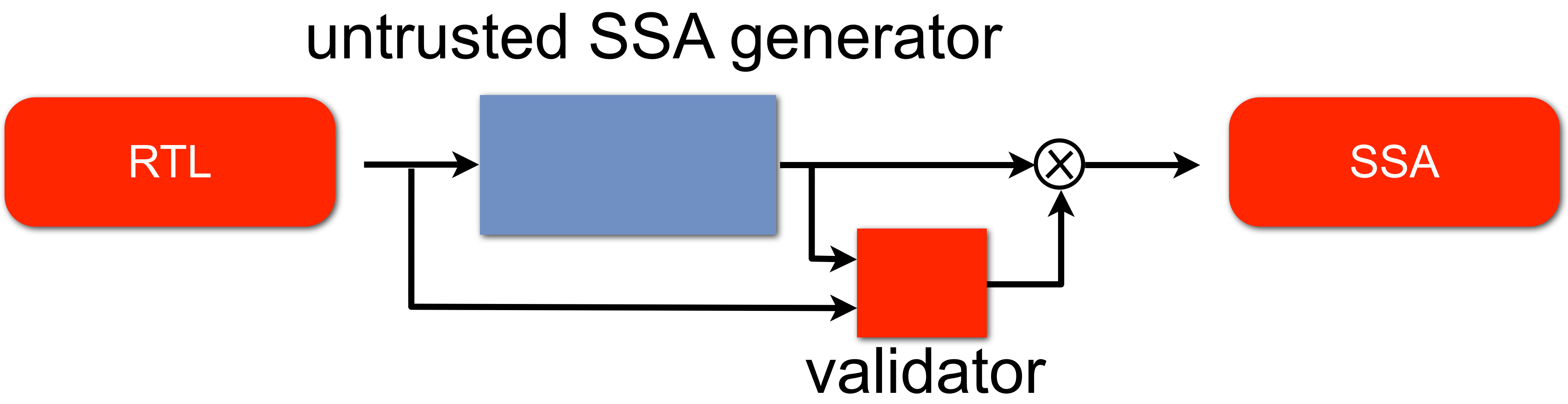
Verified translation validation

Example: SSA generation (in CompCert SSA extension)



Verified translation validation

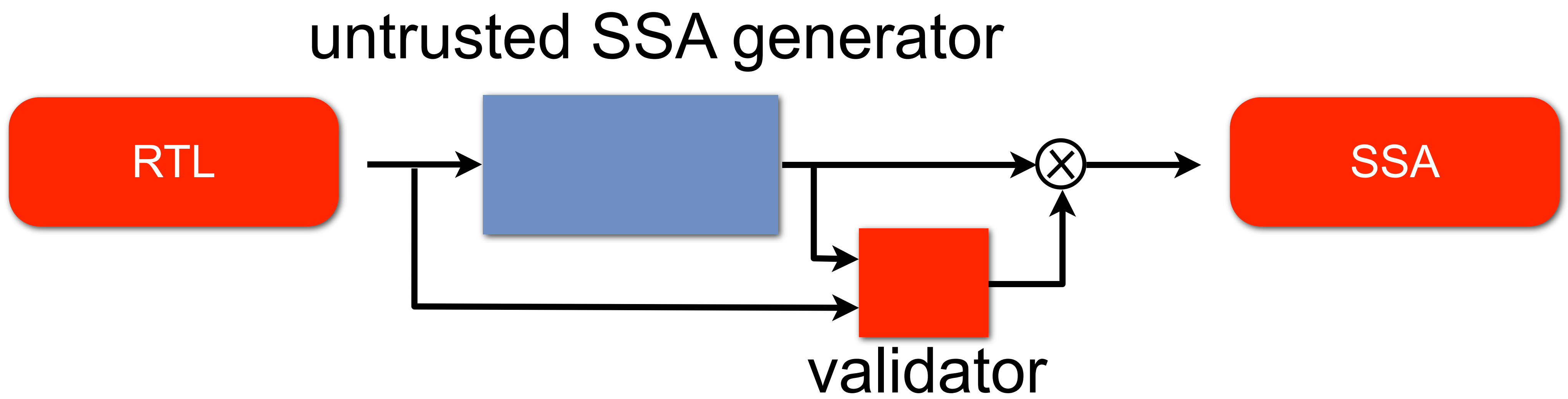
Example: SSA generation (in CompCert SSA extension)



The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan’s dominator tree construction and frontier dominance computation.

Verified translation validation

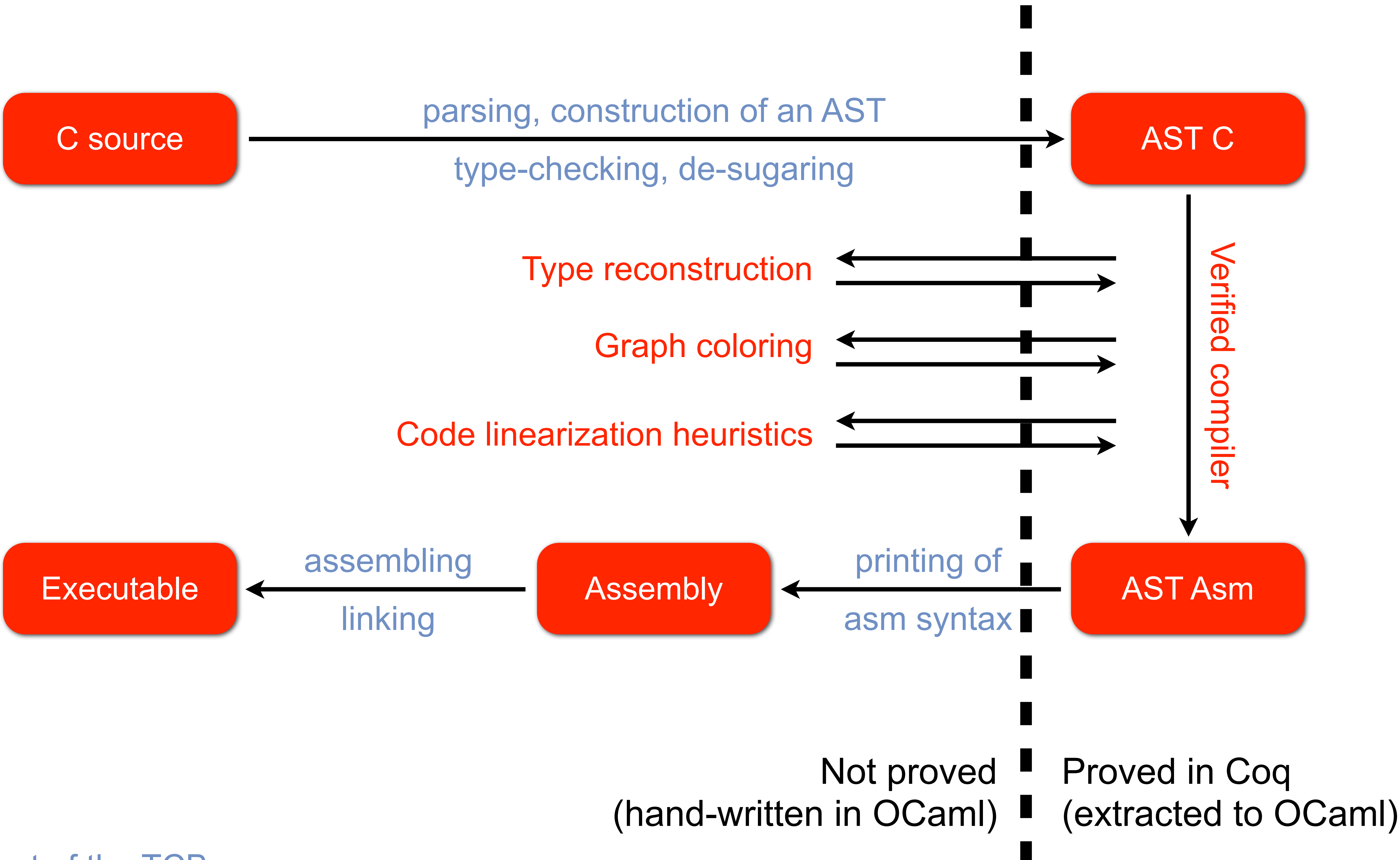
Example: SSA generation (in CompCert SSA extension)



The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan's dominator tree construction and frontier dominance computation.

We prove the validator is *complete* with respect to this family of algorithms.

The whole CompCert compiler

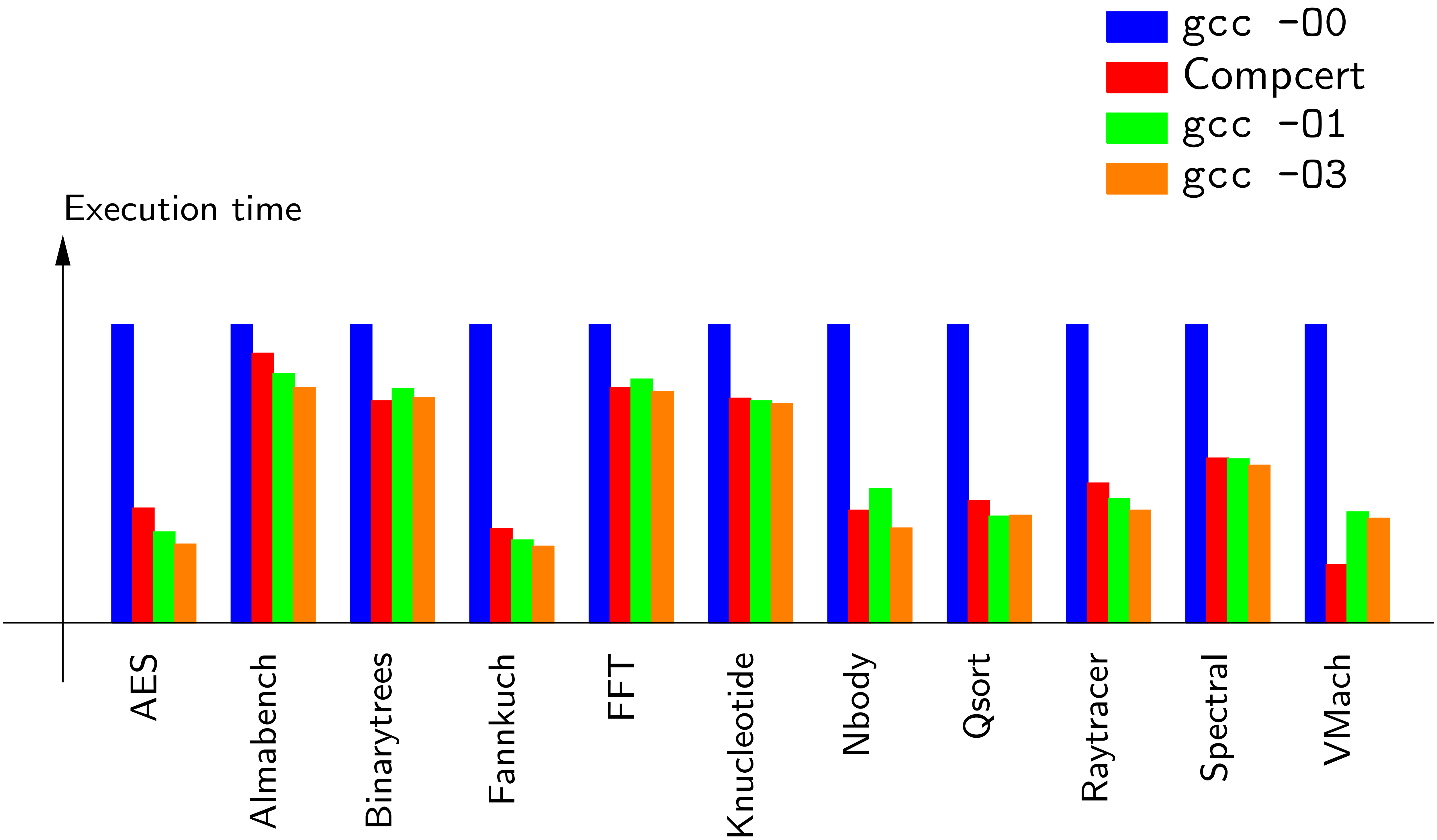


Part of the TCB

Not part of the TCB

Performance of generated code

(On a PowerPC G5 processor)



Conclusions

The formal verification of realistic compilers is feasible.

(Within the limitations of contemporary proof assistants)

Much work remains:

- Shrinking the TCB
  (e.g. verified parsing, validated assembling & linking).
- More optimizations
  (see CompCert SSA).
- Front-ends for other languages
- Concurrency
  (see Sevcik et al's CompCert TSO and Appel and al's Verified Software Toolchain).
- Connections with source-level verification
  (ongoing french project on a *verified* C static analyzer)

Formal Verification of a C Value Analysis

Sandrine Blazy, Vincent Laporte, Andre Maroneze, and David Pichardie,
to be presented at the 20th Static Analysis Symposium 2013

This work is part of the common Verasco project between

Airbus

INRIA Paris Rocquencourt (Gallium, Abstraction)

INRIA Saclay (Toccata)

Université Rennes I (Celtique)

VERIMAG

VERASCO

http://verasco.imag.fr

Why a value analysis for CompCert ?

CompCert provides strong guarantees but only for programs with well behaved behaviors

Compiler Theorem

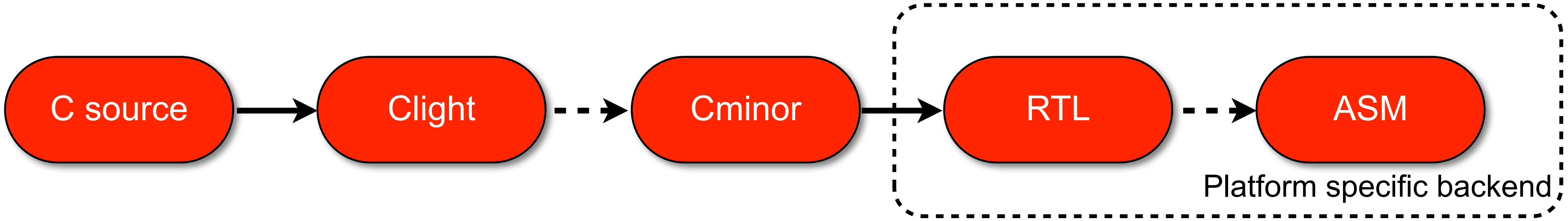
```
Theorem transf_c_program_is_refinement:
  ∀ p tp,
  transf_c_program p = OK tp →
  (∀ beh, exec_C_program p beh → not_wrong beh) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

A powerful and verified static analysis aims at proving that a program only exhibits well behaved behaviors

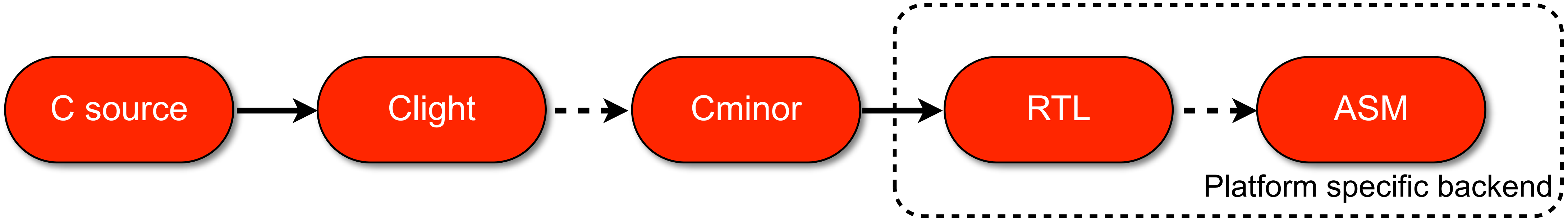
Analyser Theorem

```
Theorem analyzer_is_correct:
  ∀ p,
  analyzer p = Success →
  (∀ beh, exec_C_program p beh → not_wrong beh).
```

Which CompCert representation ?



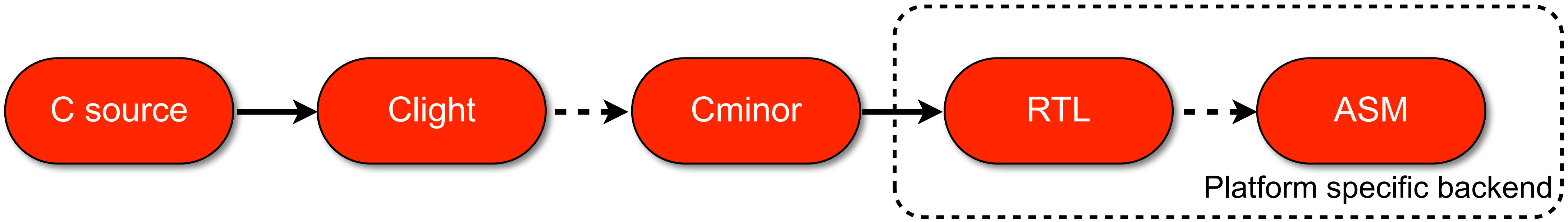
Which CompCert representation ?



C source?

- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language)

Which CompCert representation ?



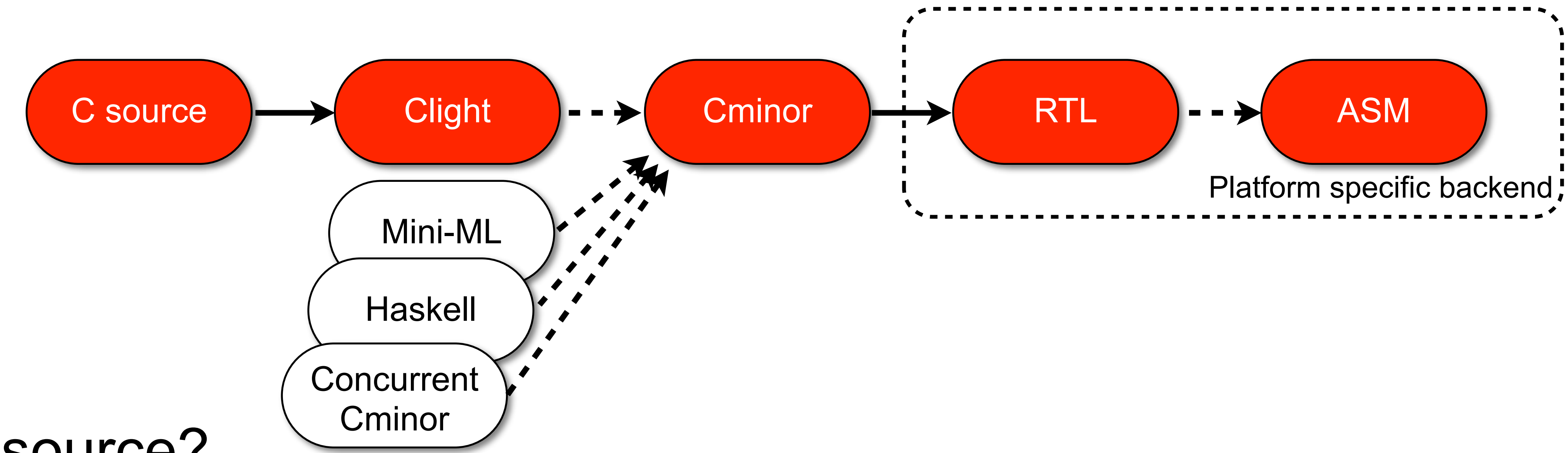
C source?

- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language)

RTL?

- the place where most CompCert static analyses take place
- but platform specific, flat expressions

Which CompCert representation ?



C source?

- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language)

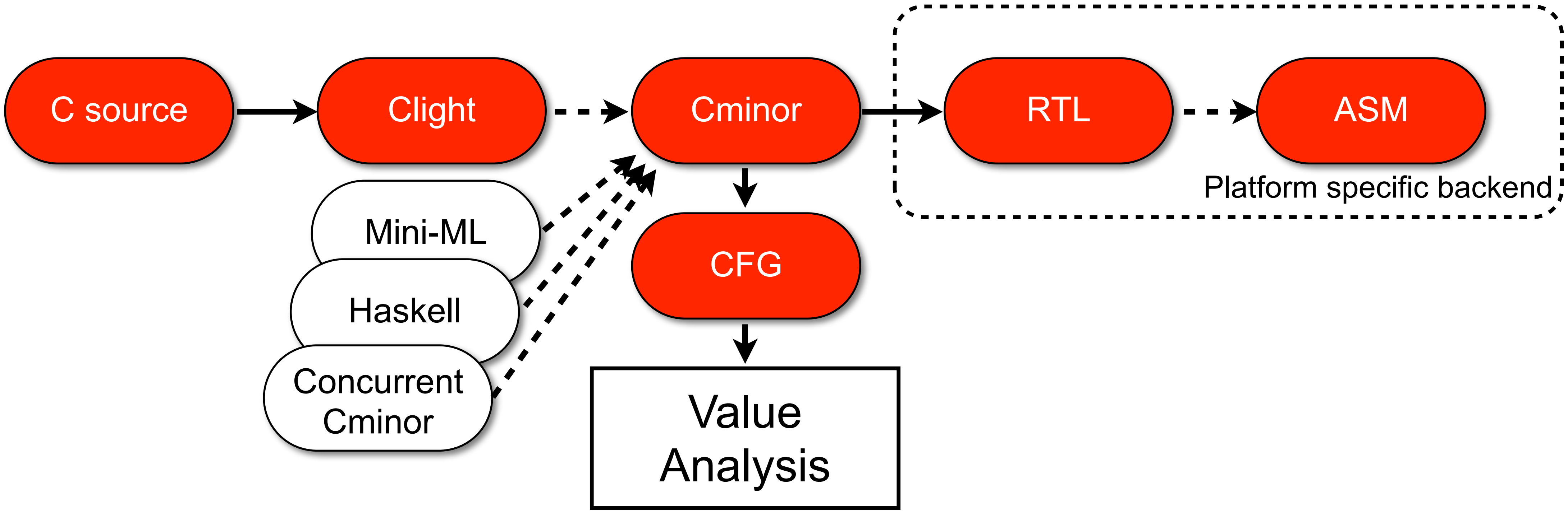
RTL?

- the place where most CompCert static analyses take place
- but platform specific, flat expressions

Cminor?

- the last step before platform specific semantics
- designed to welcome forthcoming extensions
- but control flow still less uniform than in RTL (nested blocks and exits)

Which CompCert representation ?



CFG (Control Flow Graph)

- a new representation recently added by JH. Jourdan and Xavier Leroy
- Cminor expressions (i.e., side-effect free C expressions)
- control flow graphs with explicit program points
- control flow is restricted to simple unconditional and conditional jumps
- platform independent

CFG syntax

Statements:	$i ::=$	$\texttt{skip}(l)$	no operation (go to l)
	$\mid$	$\texttt{assign}(id, a, l)$	assignment
	$\mid$	$\texttt{store}(\kappa, a, a, l)$	memory store
	$\mid$	$\texttt{if}(a, l_{true}, l_{false})$	if statement
	$\mid$	$\texttt{call}(sig, id^?, a, a*, l)$	function call
	$\mid$	$\texttt{return}(a)^?$	function return

CFG syntax

Statements:	$i ::=$	$\texttt{skip}(l)$	no operation (go to l)
	$\mid$	$\texttt{assign}(id, a, l)$	assignment
	$\mid$	$\texttt{store}(\kappa, a, a, l)$	memory store
	$\mid$	$\texttt{if}(a, l_{true}, l_{false})$	if statement
	$\mid$	$\texttt{call}(sig, id^?, a, a*, l)$	function call
	$\mid$	$\texttt{return}(a)^?$	function return
Expressions:	$a ::=$	id	variable identifier
	$\mid$	c	constant
	$\mid$	$op_1 \ a$	unary arithmetic operation
	$\mid$	$a_1 \ op_2 \ a_2$	binary arithmetic operation
	$\mid$	$a_1 ? \ a_2 : a_3$	conditional expression
	$\mid$	$\texttt{load}(\kappa, a)$	memory load

CFG syntax

Constants:	$c ::= n \mid f$	integer and floating-point constants
	$\mid \texttt{addrsymbol}(id, n)$	address of a symbol plus an offset
	$\mid \texttt{addrstack}(n)$	stack pointer plus a given offset
Unary op.:	$op_1 ::= \texttt{cast8unsigned}$	8-bit zero extension
	$\mid \texttt{cast8signed}$	8-bit sign extension
	$\mid \texttt{cast16unsigned}$	16-bit zero extension
	$\mid \texttt{cast16signed}$	16-bit sign extension
	$\mid \texttt{boolval}$	0 if null, 1 if non-null
	$\mid \texttt{negint}$	integer opposite
	$\mid \texttt{notbool}$	boolean negation
	$\mid \texttt{notint}$	bitwise complement
Binary op.:	$op_2 ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	arithmetic integer operators
	$\mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\textbar} \mid \texttt{\^{}}$	bitwise operators
	$\mid \texttt{/}_u \mid \texttt{\%}_u \mid \texttt{>>}_u$	unsigned operators
	$\mid \texttt{cmp}(b)$	integer signed comparisons
	$\mid \texttt{cmpu}(b)$	integer unsigned comparisons
Comparisons:	$b ::= \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=}$	relational operators

CFG semantics

A CFG program manipulate values

- `Vint i` with `i` an integer
- `Vptr b i` with a memory block identifier and `i` an offset (integer)
- `Vfloat f` with `f` a floating-point number
- `Vundef` (contents of uninitialized memory)

An integer is modelled with a dependent record

```
Definition wordsize: nat := 32.
Definition modulus : Z := two_power_nat wordsize.
Definition half_modulus : Z := modulus / 2.
Definition max_unsigned : Z := modulus - 1.
Definition max_signed : Z := half_modulus - 1.
Definition min_signed : Z := - half_modulus.

Record int := { intval: Z; intrange: 0 <= intval < modulus }.
```

and a signed and an unsigned interpretation.

CFG semantics

A small step semantics `step: state -> trace -> state -> Prop` models program execution (with a trace of behavior)

```
Inductive step: state → trace → state → Prop :=
  | step_skip:
      ∀ s f sp pc e m pc′,
      (fn_code f)!pc = Some(Iskip pc′) →
      step (State s f sp pc e m)
        E0 (State s f sp pc′ e m)
  | step_assign:
      ∀ s f sp pc e m id a pc′ v,
      (fn_code f)!pc = Some(Iassign id a pc′) →
      eval_expr ge sp e m a v →
      step (State s f sp pc e m)
        E0 (State s f sp pc′ (PTree.set id v e) m)
  | step_store:
      ∀ s f sp pc e m chunk addr src pc′ vaddr vsrc m′,
      (fn_code f)!pc = Some(Istore chunk addr src pc′) →
      eval_expr ge sp e m addr vaddr →
      eval_expr ge sp e m src vsrc →
      Mem.storev chunk m vaddr vsrc = Some m′ →
      step (State s f sp pc e m)
        E0 (State s f sp pc′ e m′)
  [...]
```

CFG Analyser

For all reachable states (according to `step`), for all all local variables that contain a value (Vint i) or (Vptr b i), the analyser computes a range for the integer i.

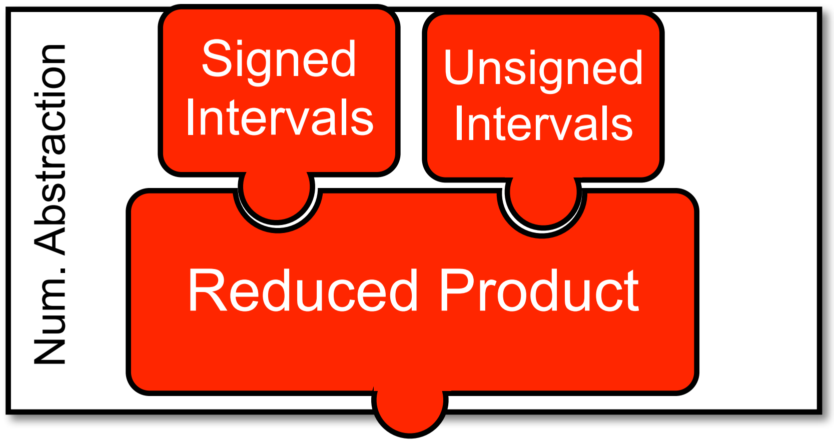
In the future, these inferred properties will be used to prove execution safety.

Abstractions currently handled

- interval abstraction for signed and unsigned interpretation of integers
- each local variable is abstracted by a pair of intervals.

But the main contribution of the current work is a set of Coq interfaces

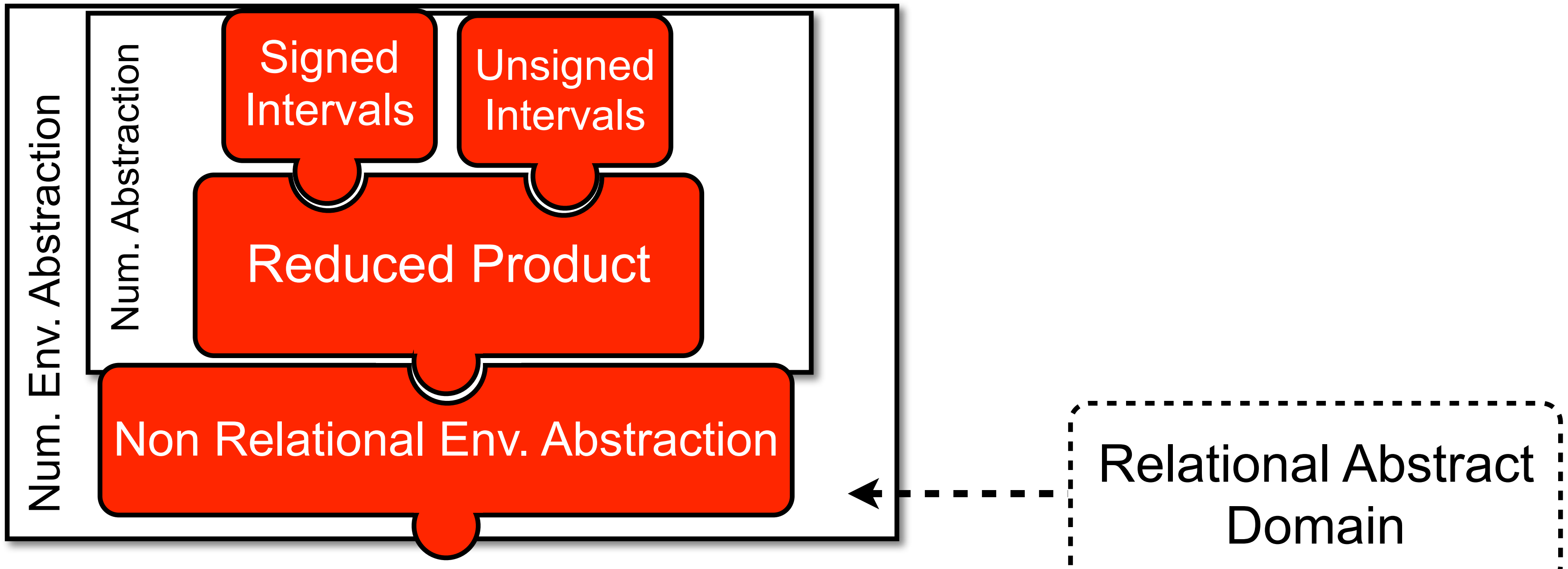
Analyser Interfaces



Numeric Abstraction

- abstraction of a single integer value,
- takes into account all the CompCert numerical operations
- reduced product: combine two abstractions for better precision
  Example:
  if $\mathrm{signed}(i) \in [-256, 255] \ \wedge \ \mathrm{unsigned}(i) \in [512, 2^{32}-1]$
  then $\mathrm{signed}(i) \in [-256, -1] \ \wedge \ \mathrm{unsigned}(i) \in [2^{32}-256, 2^{32}-1]$

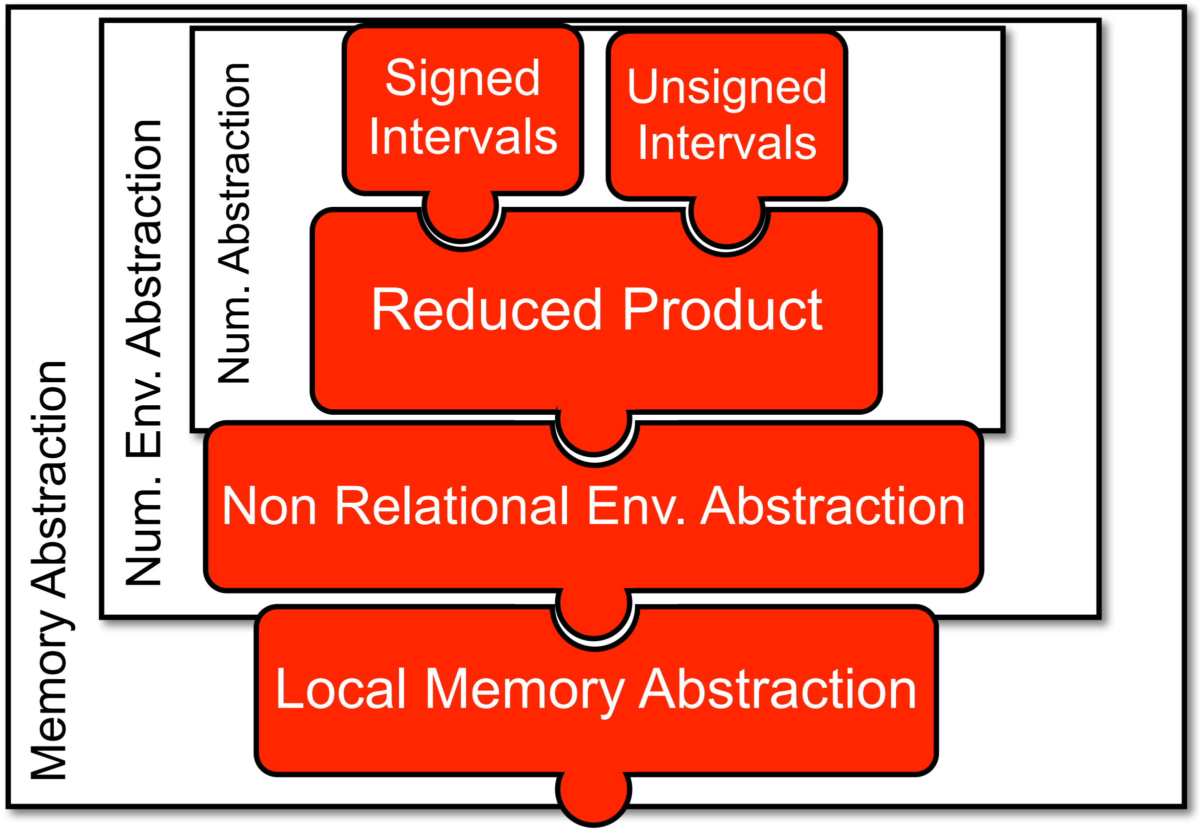
Analyser Interfaces



Numeric Environment Abstraction

- parameterised by an abstract notion of variable
- instantiated with a non-relational abstraction (each variable is given a numerical abstraction)
- interface ready for relational abstraction

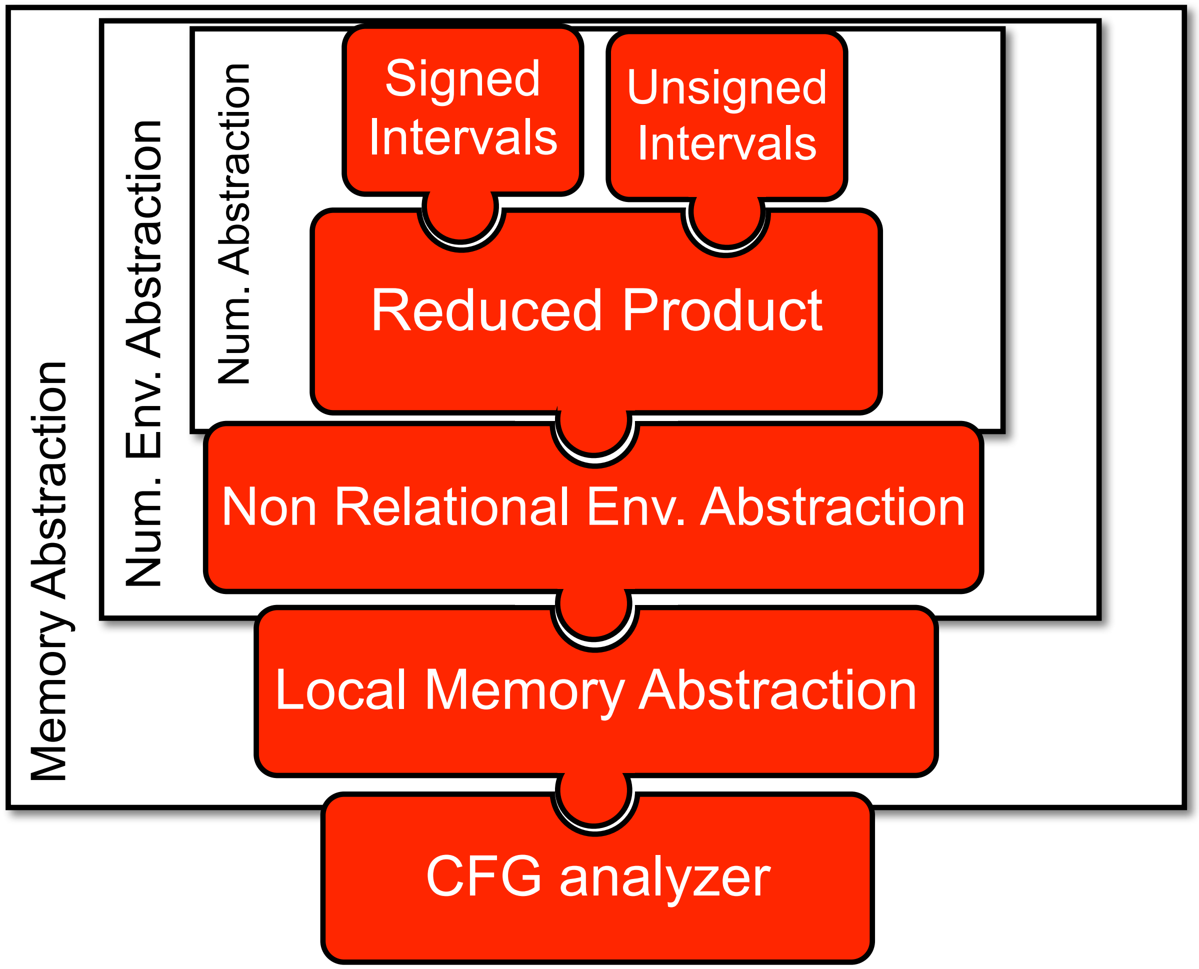
Analyser Interfaces



Memory Abstraction

- the only signature where the C memory is exposed
- currently implemented by mapping V with local variables
- ready for more ambitious abstractions where V is also mapped to memory cells

Analyser Interfaces



Generic analyzer

- parameterised by any memory abstraction
- CFG program are unstructured: need to build widening strategies on unstructured control flow graph
  => we let an external tool computes a post-fixpoint and check the result in Coq

Conclusions

A first step toward a verified C verifier

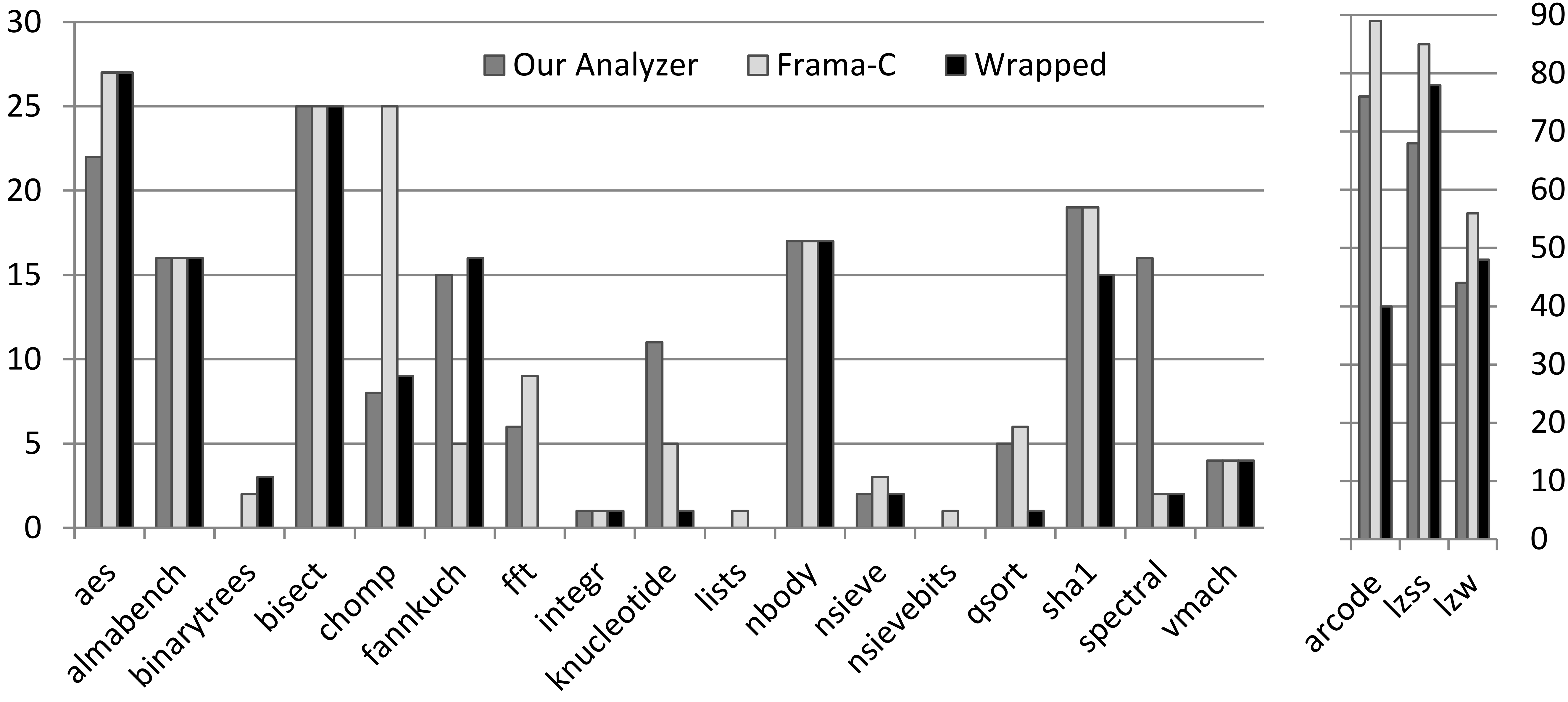
- experimental evaluations show that our tool compare already well with Frama-C value analysis (more in the paper)

Next steps

- abstract interpretation a source level
- relational abstract domains
- abstract domains for floating-point numbers

Experiments on safety critical programs

- stress test the efficiency of the analyser
- add new abstract domains for specific program patterns


Our Analyzer
Frama-C
Wrapped
30
25
20
15
10
5
0
90
80
70
60
50
40
30
20
10
0
aes
almabench
binarytrees
bisect
chomp
fannkuch
fft
integr
knucleotide
lists
nbody
nsieve
nsievebits
qsort
sha1
spectral
vmach
arcode
lzss
lzw