The CompCert verified C compiler

Compiler built and proved by Xavier Leroy et al.

Talk given by David Pichardie - Harvard University / INRIA Rennes

Slides largely inspired by Xavier Leroy own material

Critical embedded software

```
        .const
        .align  2
___stringlit_1:
        .ascii  "integr(square, 0.0, 1.0,
        .text
        .align  4
_square:
        subl    $12, %esp
        leal    16(%esp), %edx
        movl    %edx, 0(%esp)
        movl    0(%esp), %edx
        movsd   0(%edx), %xmm0
        mulsd   %xmm0, %xmm0
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        addl    $12, %esp
        ret
        .text
        .align  4
_integr:
        subl    $60, %esp
        leal    64(%esp), %edx
        movl    %edx, 8(%esp)
        movl    %ebx, 20(%esp)
        movl    %esi, 24(%esp)
        movl    8(%esp), %edx
        movl    0(%edx), %esi
        movl    8(%esp), %edx
        movsd   4(%edx), %xmm6
        movsd   %xmm6, 48(%esp)
        movl    8(%esp), %edx
        movsd   12(%edx), %xmm6
        movsd   %xmm6, 32(%esp)
        movl    8(%esp), %edx
        movl    20(%edx), %ebx
        movsd   32(%esp), %xmm6
        movsd   48(%esp), %xmm7
        subsd   %xmm7, %xmm6
        movsd   %xmm6, 32(%esp)
        cvtsi2sd %ebx, %xmm1
        movsd   32(%esp), %xmm6
        divsd   %xmm1, %xmm6
        movsd   %xmm6, 32(%esp)
        xorpd   %xmm6, %xmm6 # +0.0
        movsd   %xmm6, 40(%esp)
L100:
        cmpl    $0, %ebx
        jle     L101
        movsd   48(%esp), %xmm6
        movsd   %xmm6, 0(%esp)
        call    *%esi
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        movsd   40(%esp), %xmm6
        addsd   %xmm0, %xmm6
        movsd   %xmm6, 40(%esp)
        leal    -1(%ebx), %ebx
        movsd   48(%esp), %xmm6
        movsd   32(%esp), %xmm7
        addsd   %xmm7, %xmm6
        movsd   %xmm6, 48(%esp)
        jmp     L100
L101:
        movsd   40(%esp), %xmm6
        movsd   32(%esp), %xmm7
        mulsd   %xmm7, %xmm6
        movsd   %xmm6, 40(%esp)
        fldl    40(%esp)
        movl    20(%esp), %ebx
        movl    24(%esp), %esi
        addl    $60, %esp
        ret
        .text
        .align  4
        .globl _test
_test:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 24(%esp)
        movl    %ebx, 36(%esp)
        movl    24(%esp), %edx
        movl    0(%edx), %eax
        leal    _square, %ebx
        xorpd   %xmm2, %xmm2 # +0.0
        movsd   L102, %xmm1 # 1
        movl    %ebx, 0(%esp)
        movsd   %xmm2, 4(%esp)
        movsd   %xmm1, 12(%esp)
        movl    %eax, 20(%esp)
        call    _integr
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        movl    36(%esp), %ebx
        addl    $44, %esp
        ret
        .const_data
        .align  3
L102:   .quad   0x3ff0000000000000
        .text
        .align  4
        .globl _main
_main:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 16(%esp)
        movl    %ebx, 28(%esp)
        movl    16(%esp), %edx
        movl    0(%edx), %ebx
        movl    16(%esp), %edx
        movl    4(%edx), %eax
        cmpl    $2, %ebx
        jge     L103
        movl    $10000000, %ebx
        jmp     L104
L103:
        movl    4(%eax), %eax
        movl    %eax, 0(%esp)
        call    _atoi
        movl    %eax, %ebx
L104:
        movl    %ebx, 0(%esp)
        call    _test
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        leal    ___stringlit_1, %eax
        movl    %eax, 0(%esp)
        movl    %ebx, 4(%esp)
        movsd   %xmm0, 8(%esp)
        call    _printf
        xorl    %eax, %eax
        movl    28(%esp), %ebx
        addl    $44, %esp
        ret
```

critical_prog.ppc

Critical embedded software

High degree of assurance is required

- is the program critical_prog.ppc safe ?

```
        .const
        .align  2
___stringlit_1:
        .ascii  "integr(square, 0.0, 1.0,
        .text
        .align  4
_square:
        subl    $12, %esp
        leal    16(%esp), %edx
        movl    %edx, 0(%esp)
        movl    0(%esp), %edx
        movsd   0(%edx), %xmm0
        mulsd   %xmm0, %xmm0
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        addl    $12, %esp
        ret
        .text
        .align  4
_integr:
        subl    $60, %esp
        leal    64(%esp), %edx
        movl    %edx, 8(%esp)
        movl    %ebx, 20(%esp)
        movl    %esi, 24(%esp)
        movl    8(%esp), %edx
        movl    0(%edx), %esi
        movl    8(%esp), %edx
        movsd   4(%edx), %xmm6
        movsd   %xmm6, 48(%esp)
        movl    8(%esp), %edx
        movsd   12(%edx), %xmm6
        movsd   %xmm6, 32(%esp)
        movl    8(%esp), %edx
        movl    20(%edx), %ebx
        movsd   32(%esp), %xmm6
        movsd   48(%esp), %xmm7
        subsd   %xmm7, %xmm6
        movsd   %xmm6, 32(%esp)
        cvtsi2sd %ebx, %xmm1
        movsd   32(%esp), %xmm6
        divsd   %xmm1, %xmm6
        movsd   %xmm6, 32(%esp)
        xorpd   %xmm6, %xmm6 # +0.0
        movsd   %xmm6, 40(%esp)
L100:
        cmpl    $0, %ebx
        jle     L101
        movsd   48(%esp), %xmm6
        movsd   %xmm6, 0(%esp)
        call    *%esi
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        movsd   40(%esp), %xmm6
        addsd   %xmm0, %xmm6
        movsd   %xmm6, 40(%esp)
        leal    -1(%ebx), %ebx
        movsd   48(%esp), %xmm6
        movsd   32(%esp), %xmm7
        addsd   %xmm7, %xmm6
        movsd   %xmm6, 48(%esp)
        jmp     L100
L101:
        movsd   40(%esp), %xmm6
        movsd   32(%esp), %xmm7
        mulsd   %xmm7, %xmm6
        movsd   %xmm6, 40(%esp)
        fldl    40(%esp)
        movl    20(%esp), %ebx
        movl    24(%esp), %esi
        addl    $60, %esp
        ret
        .text
        .align  4
        .globl _test
_test:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 24(%esp)
        movl    %ebx, 36(%esp)
        movl    24(%esp), %edx
        movl    0(%edx), %eax
        leal    _square, %ebx
        xorpd   %xmm2, %xmm2 # +0.0
        movsd   L102, %xmm1 # 1
        movl    %ebx, 0(%esp)
        movsd   %xmm2, 4(%esp)
        movsd   %xmm1, 12(%esp)
        movl    %eax, 20(%esp)
        call    _integr
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        movl    36(%esp), %ebx
        addl    $44, %esp
        ret
        .const_data
        .align  3
L102:   .quad   0x3ff0000000000000
        .text
        .align  4
        .globl _main
_main:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 16(%esp)
        movl    %ebx, 28(%esp)
        movl    16(%esp), %edx
        movl    0(%edx), %ebx
        movl    16(%esp), %edx
        movl    4(%edx), %eax
        cmpl    $2, %ebx
        jge     L103
        movl    $10000000, %ebx
        jmp     L104
L103:
        movl    4(%eax), %eax
        movl    %eax, 0(%esp)
        call    _atoi
        movl    %eax, %ebx
L104:
        movl    %ebx, 0(%esp)
        call    _test
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        leal    ___stringlit_1, %eax
        movl    %eax, 0(%esp)
        movl    %ebx, 4(%esp)
        movsd   %xmm0, 8(%esp)
        call    _printf
        xorl    %eax, %eax
        movl    28(%esp), %ebx
        addl    $44, %esp
        ret
```

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)

```
        .const
        .align  2
___stringlit_1:
        .ascii  "integr(square, 0.0, 1.0,
        .text
        .align  4
_square:
        subl    $12, %esp
        leal    16(%esp), %edx
        movl    %edx, 0(%esp)
        movl    0(%esp), %edx
        movsd   0(%edx), %xmm0
        mulsd   %xmm0, %xmm0
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        addl    $12, %esp
        ret
        .text
        .align  4
_integr:
        subl    $60, %esp
        leal    64(%esp), %edx
        movl    %edx, 8(%esp)
        movl    %ebx, 20(%esp)
        movl    %esi, 24(%esp)
        movl    8(%esp), %edx
        movl    0(%edx), %esi
        movl    8(%esp), %edx
        movsd   4(%edx), %xmm6
        movsd   %xmm6, 48(%esp)
        movl    8(%esp), %edx
        movsd   12(%edx), %xmm6
        movsd   %xmm6, 32(%esp)
        movl    8(%esp), %edx
        movl    20(%edx), %ebx
        movsd   32(%esp), %xmm6
        movsd   48(%esp), %xmm7
        subsd   %xmm7, %xmm6
        movsd   %xmm6, 32(%esp)
        cvtsi2sd %ebx, %xmm1
        movsd   32(%esp), %xmm6
        divsd   %xmm1, %xmm6
        movsd   %xmm6, 32(%esp)
        xorpd   %xmm6, %xmm6 # +0.0
        movsd   %xmm6, 40(%esp)
L100:
        cmpl    $0, %ebx
        jle     L101
        movsd   48(%esp), %xmm6
        movsd   %xmm6, 0(%esp)
        call    *%esi
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        movsd   40(%esp), %xmm6
        addsd   %xmm0, %xmm6
        movsd   %xmm6, 40(%esp)
        leal    -1(%ebx), %ebx
        movsd   48(%esp), %xmm6
        movsd   32(%esp), %xmm7
        addsd   %xmm7, %xmm6
        movsd   %xmm6, 48(%esp)
        jmp     L100
L101:
        movsd   40(%esp), %xmm6
        movsd   32(%esp), %xmm7
        mulsd   %xmm7, %xmm6
        movsd   %xmm6, 40(%esp)
        fldl    40(%esp)
        movl    20(%esp), %ebx
        movl    24(%esp), %esi
        addl    $60, %esp
        ret
        .text
        .align  4
        .globl _test
_test:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 24(%esp)
        movl    %ebx, 36(%esp)
        movl    24(%esp), %edx
        movl    0(%edx), %eax
        leal    _square, %ebx
        xorpd   %xmm2, %xmm2 # +0.0
        movsd   L102, %xmm1 # 1
        movl    %ebx, 0(%esp)
        movsd   %xmm2, 4(%esp)
        movsd   %xmm1, 12(%esp)
        movl    %eax, 20(%esp)
        call    _integr
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        movl    36(%esp), %ebx
        addl    $44, %esp
        ret
        .const_data
        .align  3
L102:   .quad   0x3ff0000000000000
        .text
        .align  4
        .globl _main
_main:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 16(%esp)
        movl    %ebx, 28(%esp)
        movl    16(%esp), %edx
        movl    0(%edx), %ebx
        movl    16(%esp), %edx
        movl    4(%edx), %eax
        cmpl    $2, %ebx
        jge     L103
        movl    $10000000, %ebx
        jmp     L104
L103:
        movl    4(%eax), %eax
        movl    %eax, 0(%esp)
        call    _atoi
        movl    %eax, %ebx
L104:
        movl    %ebx, 0(%esp)
        call    _test
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        leal    ___stringlit_1, %eax
        movl    %eax, 0(%esp)
        movl    %ebx, 4(%esp)
        movsd   %xmm0, 8(%esp)
        call    _printf
        xorl    %eax, %eax
        movl    28(%esp), %ebx
        addl    $44, %esp
        ret
```

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 !

```
        .const
        .align  2
___stringlit_1:
        .ascii  "integr(square, 0.0, 1.0,
        .text
        .align  4
_square:
        subl    $12, %esp
        leal    16(%esp), %edx
        movl    %edx, 0(%esp)
        movl    0(%esp), %edx
        movsd   0(%edx), %xmm0
        mulsd   %xmm0, %xmm0
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        addl    $12, %esp
        ret
        .text
        .align  4
_integr:
        subl    $60, %esp
        leal    64(%esp), %edx
        movl    %edx, 8(%esp)
        movl    %ebx, 20(%esp)
        movl    %esi, 24(%esp)
        movl    8(%esp), %edx
        movl    0(%edx), %esi
        movl    8(%esp), %edx
        movsd   4(%edx), %xmm6
        movsd   %xmm6, 48(%esp)
        movl    8(%esp), %edx
        movsd   12(%edx), %xmm6
        movsd   %xmm6, 32(%esp)
        movl    8(%esp), %edx
        movl    20(%edx), %ebx
        movsd   32(%esp), %xmm6
        movsd   48(%esp), %xmm7
        subsd   %xmm7, %xmm6
        movsd   %xmm6, 32(%esp)
        cvtsi2sd %ebx, %xmm1
        movsd   32(%esp), %xmm6
        divsd   %xmm1, %xmm6
        movsd   %xmm6, 32(%esp)
        xorpd   %xmm6, %xmm6 # +0.0
        movsd   %xmm6, 40(%esp)
L100:
        cmpl    $0, %ebx
        jle     L101
        movsd   48(%esp), %xmm6
        movsd   %xmm6, 0(%esp)
        call    *%esi
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        movsd   40(%esp), %xmm6
        addsd   %xmm0, %xmm6
        movsd   %xmm6, 40(%esp)
        leal    -1(%ebx), %ebx
        movsd   48(%esp), %xmm6
        movsd   32(%esp), %xmm7
        addsd   %xmm7, %xmm6
        movsd   %xmm6, 48(%esp)
        jmp     L100
L101:
        movsd   40(%esp), %xmm6
        movsd   32(%esp), %xmm7
        mulsd   %xmm7, %xmm6
        movsd   %xmm6, 40(%esp)
        fldl    40(%esp)
        movl    20(%esp), %ebx
        movl    24(%esp), %esi
        addl    $60, %esp
        ret
        .text
        .align  4
        .globl _test
_test:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 24(%esp)
        movl    %ebx, 36(%esp)
        movl    24(%esp), %edx
        movl    0(%edx), %eax
        leal    _square, %ebx
        xorpd   %xmm2, %xmm2 # +0.0
        movsd   L102, %xmm1 # 1
        movl    %ebx, 0(%esp)
        movsd   %xmm2, 4(%esp)
        movsd   %xmm1, 12(%esp)
        movl    %eax, 20(%esp)
        call    _integr
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        subl    $8, %esp
        movsd   %xmm0, 0(%esp)
        fldl    0(%esp)
        addl    $8, %esp
        movl    36(%esp), %ebx
        addl    $44, %esp
        ret
        .const_data
        .align  3
L102:   .quad   0x3ff0000000000000
        .text
        .align  4
        .globl _main
_main:
        subl    $44, %esp
        leal    48(%esp), %edx
        movl    %edx, 16(%esp)
        movl    %ebx, 28(%esp)
        movl    16(%esp), %edx
        movl    0(%edx), %ebx
        movl    16(%esp), %edx
        movl    4(%edx), %eax
        cmpl    $2, %ebx
        jge     L103
        movl    $10000000, %ebx
        jmp     L104
L103:
        movl    4(%eax), %eax
        movl    %eax, 0(%esp)
        call    _atoi
        movl    %eax, %ebx
L104:
        movl    %ebx, 0(%esp)
        call    _test
        subl    $8, %esp
        fstpl   0(%esp)
        movsd   0(%esp), %xmm0
        addl    $8, %esp
        leal    ___stringlit_1, %eax
        movl    %eax, 0(%esp)
        movl    %ebx, 4(%esp)
        movsd   %xmm0, 8(%esp)
        call    _printf
        xorl    %eax, %eax
        movl    28(%esp), %ebx
        addl    $44, %esp
        ret
```

critical_prog.ppc

Miscompilation happens

Miscompilation happens

NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

Miscompilation happens

NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.

E. Eide & J. Regehr, EMSOFT 2008

Miscompilation happens

NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.

E. Eide & J. Regehr, EMSOFT 2008

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 290 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.

J. Regehr et al, 2010

How do we verify a compiler ?

How do we verify a compiler ?

A simple idea:

How do we verify a compiler ?

A simple idea:

Program and prove your compiler in the same language !

How do we verify a compiler ?

A simple idea:

Program and prove your compiler in the same language !

Which language ?

How do we verify a compiler ?

A simple idea:

Program and prove your compiler in the same language !

Which language ?

Coq: an animal with two faces

Coq: an animal with two faces

First face:

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system

  example:

```
sort: ∀ l: list int, { l': list int |
                          Sorted l' ∧ PermutationOf l l' }
```

Coq: an animal with two faces

First face:

- a proof assistant that allows to interactively build proof in constructive logic

Second face:

- a functional programming language with a very rich type system

  example:

```
sort: ∀ l: list int, { l': list int |
                                  Sorted l' ∧ PermutationOf l l' }
```

- with an extraction mechanism to Ocaml

```
sort: int list → int list
```

Verifying a compiler in Coq

A simple recipe

Logical Framework
(here Coq)

Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

```
Definition compiler
  (p:source) : option target := ...
```

Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

```
Definition compiler
  (p:source) : option target := ...
```

State its correctness wrt. a formal specification of the language semantics

```
Theorem compiler_is_correct :
  ∀ p, compiler p = Some p' →
       behaviors(p') ⊆ behaviors(p)
```

If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.

Compiler

Source & Target Language Semantics

Logical Framework
(here Coq)

Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

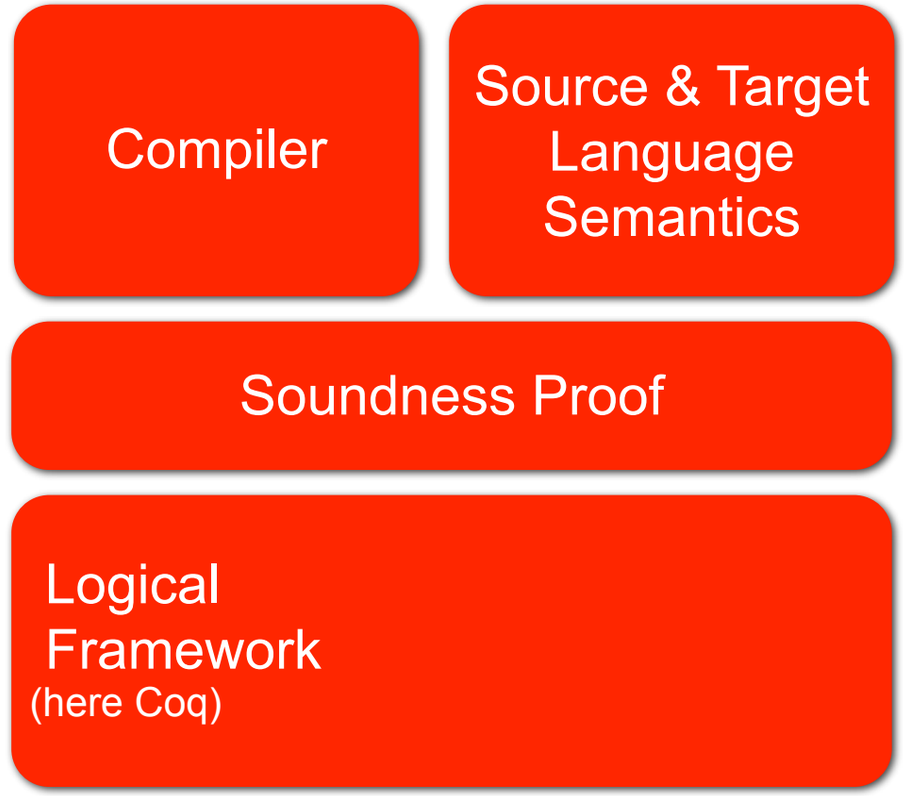
```
Definition compiler
  (p:source) : option target := ...
```

State its correctness wrt. a formal specification of the language semantics

```
Theorem compiler_is_correct :
  ∀ p, compiler p = Some p' →
       behaviors(p') ⊆ behaviors(p)
```

We interactively and mechanically prove this theorem

```
Proof. ... (* few days later *) ... Qed.
```



Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

```
Definition compiler
  (p:source) : option target := ...
```

State its correctness wrt. a formal specification of the language semantics

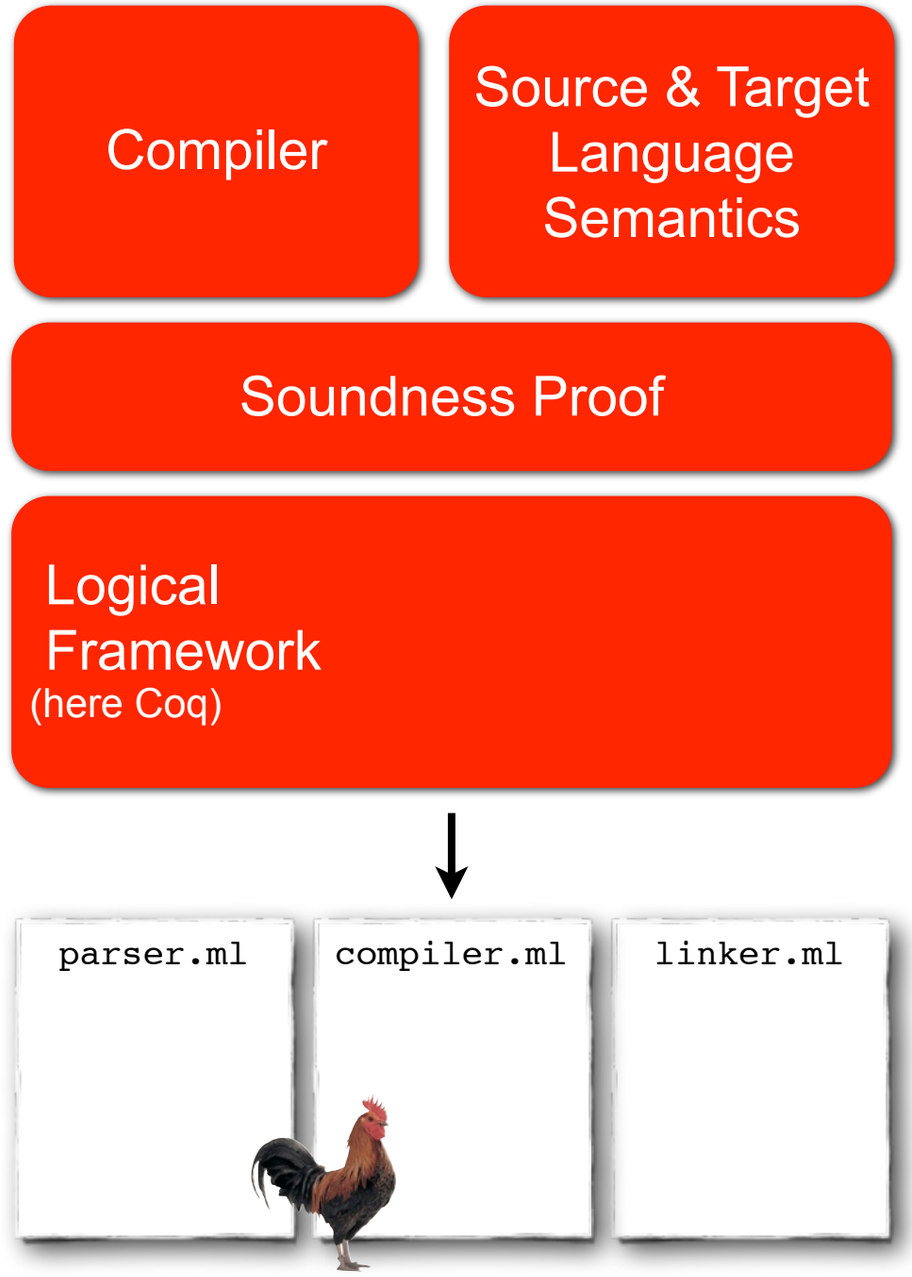
```
Theorem compiler_is_correct :
  ∀ p, compiler p = Some p' →
       behaviors(p') ⊆ behaviors(p)
```

We interactively and mechanically prove this theorem

```
Proof. ... (* few days later *) ... Qed.
```

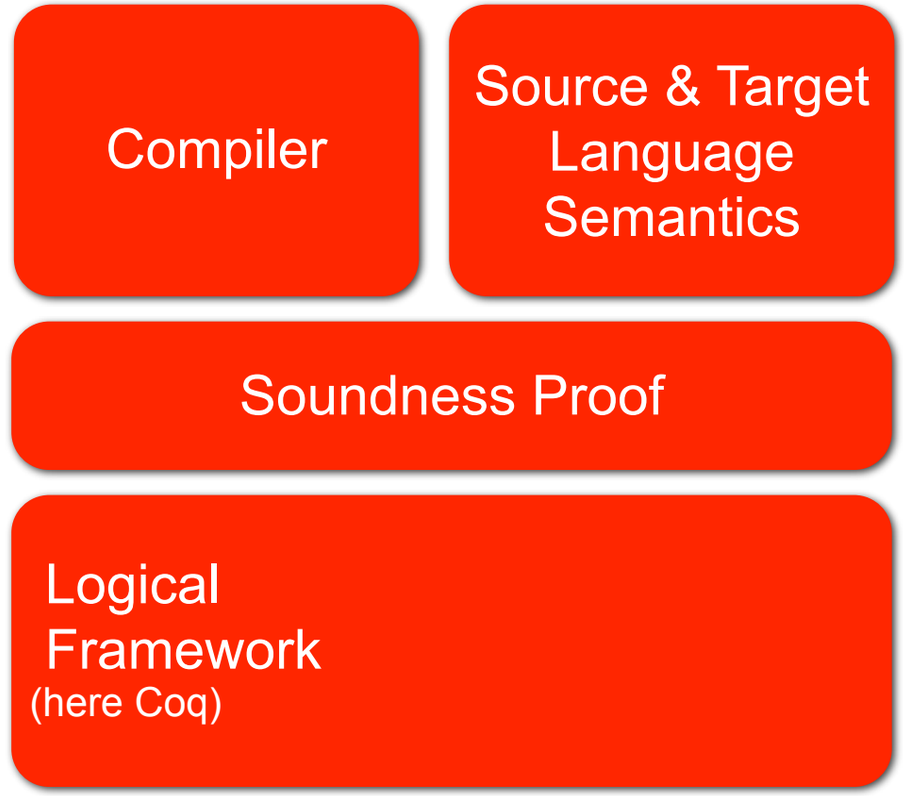
We extract an OCaml implementation of the compiler

```
Extraction compiler.
```



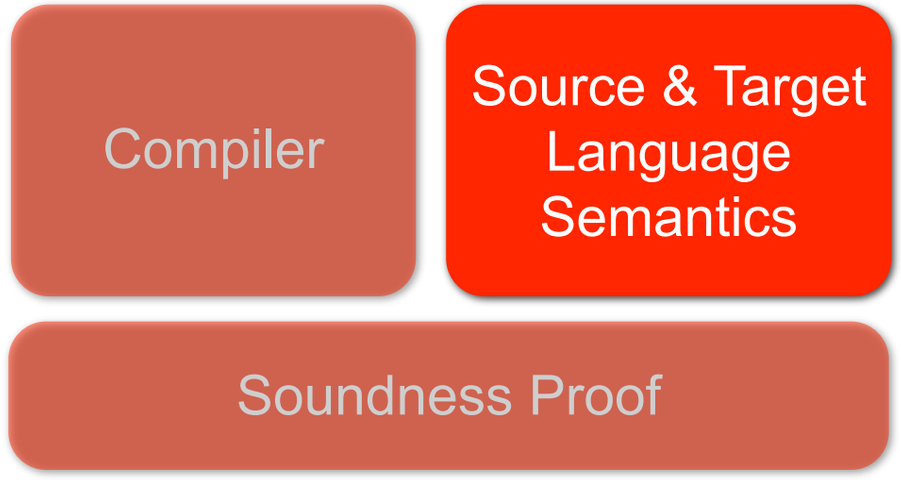
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   - already (informally) shared between any end-user programmer, compiler, static analyzer
   - must itself be rigorously proofread/animated/tested
2. Logical Framework
   - only the proof checker needs to be trusted
   - we don’t trust sophisticated decision procedures

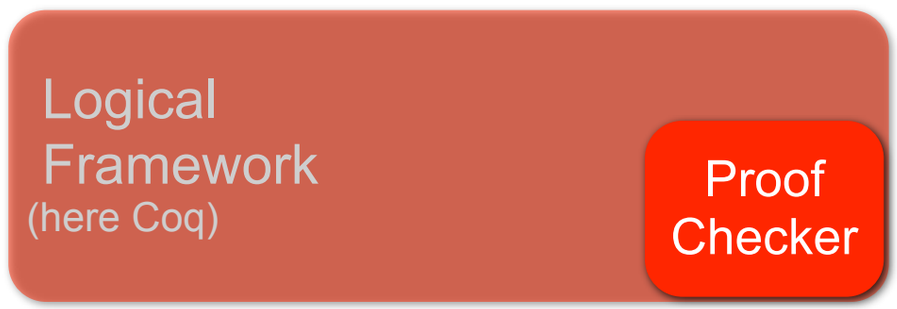


Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   - already (informally) shared between any end-user programmer, compiler, static analyzer
   - must itself be rigorously proofread/animated/tested

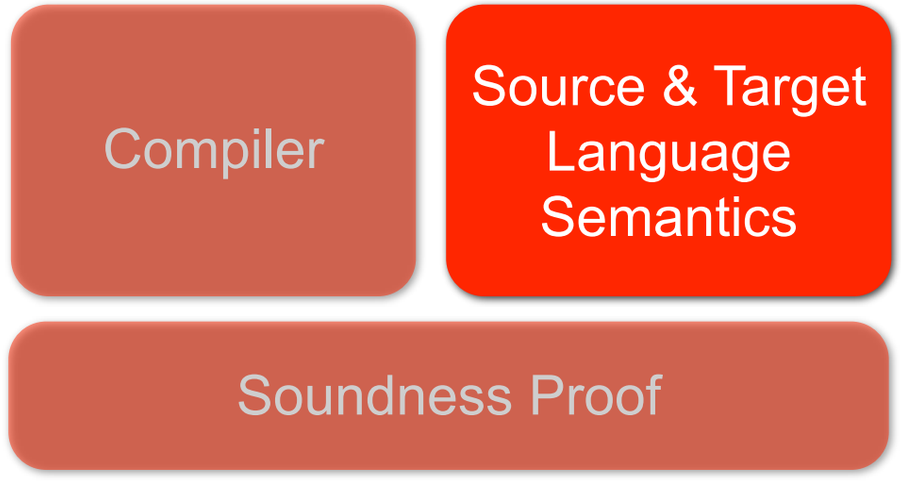


2. Logical Framework
   - only the proof checker needs to be trusted
   - we don’t trust sophisticated decision procedures

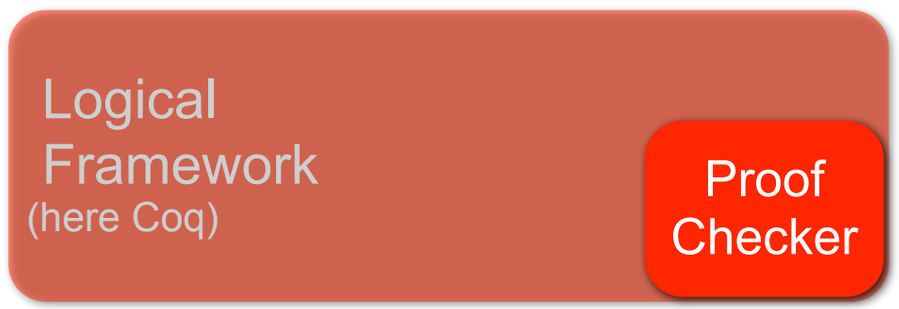


Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   - already (informally) shared between any end-user programmer, compiler, static analyzer
   - must itself be rigorously proofread/animated/tested



2. Logical Framework
   - only the proof checker needs to be trusted
   - we don't trust sophisticated decision procedures



Still a large code base but at least a *foundational* code base: logic & semantics

The CompCert project

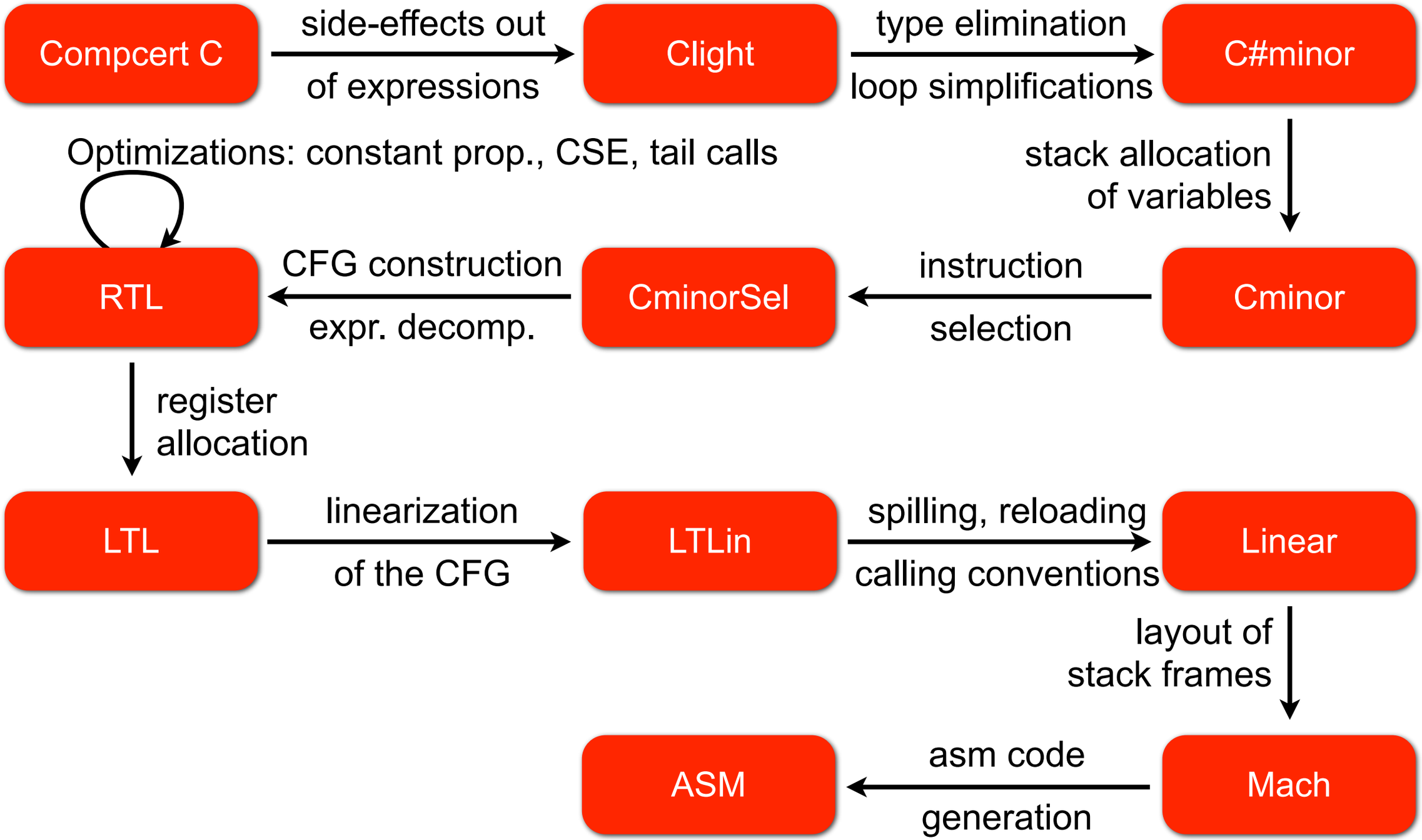
X. Leroy, S. Blazy et al.

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 optimizations.

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

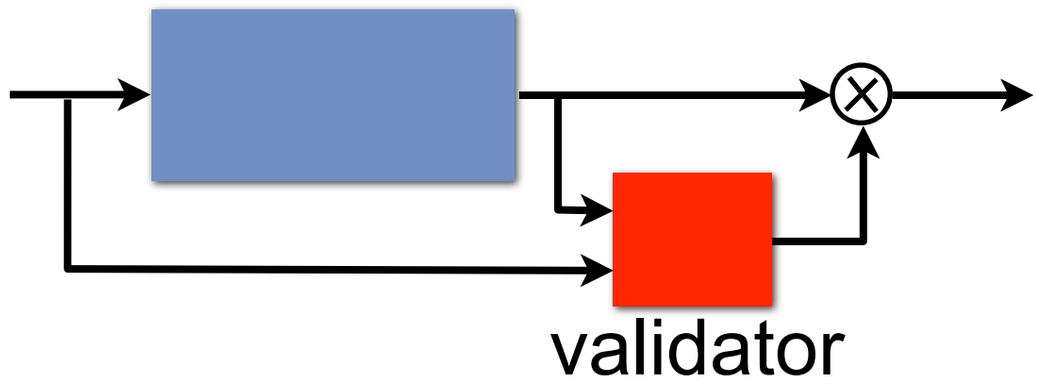
(for each pass)

Verified transformation

transformation

Verified translation validation

transformation



= formally verified

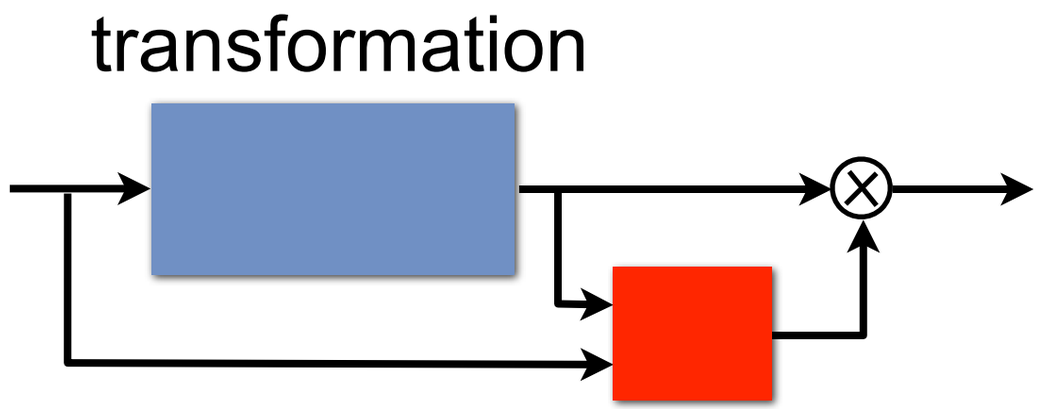
= not verified

Compiler verification patterns

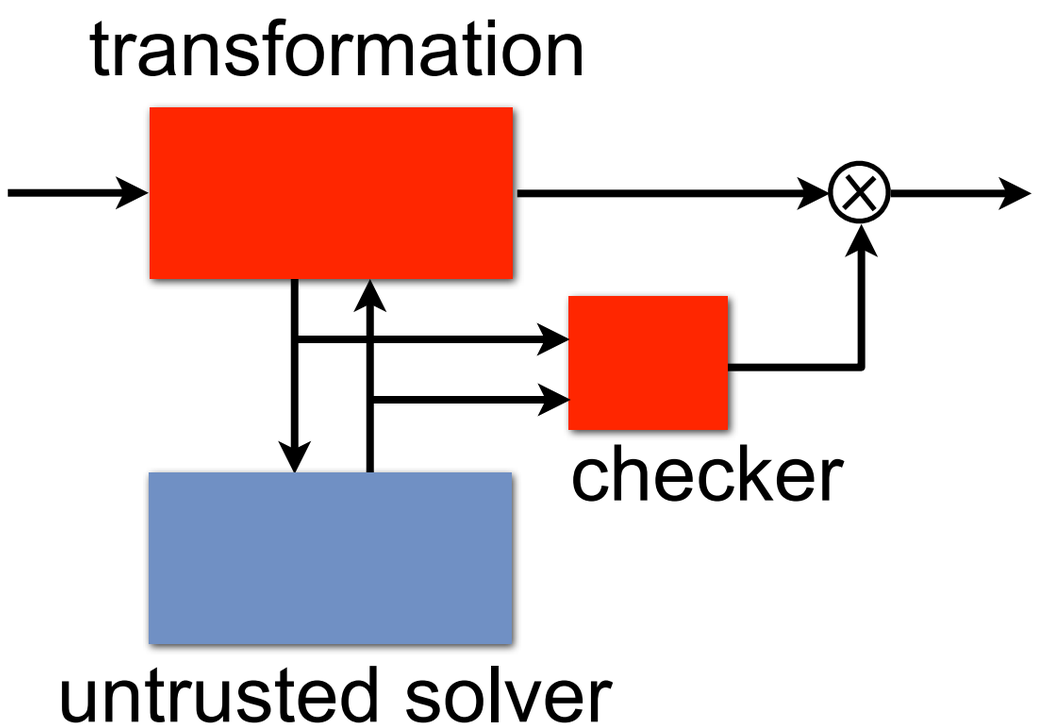
(for each pass)

Verified transformation

Verified translation validation



External solver with verified validation

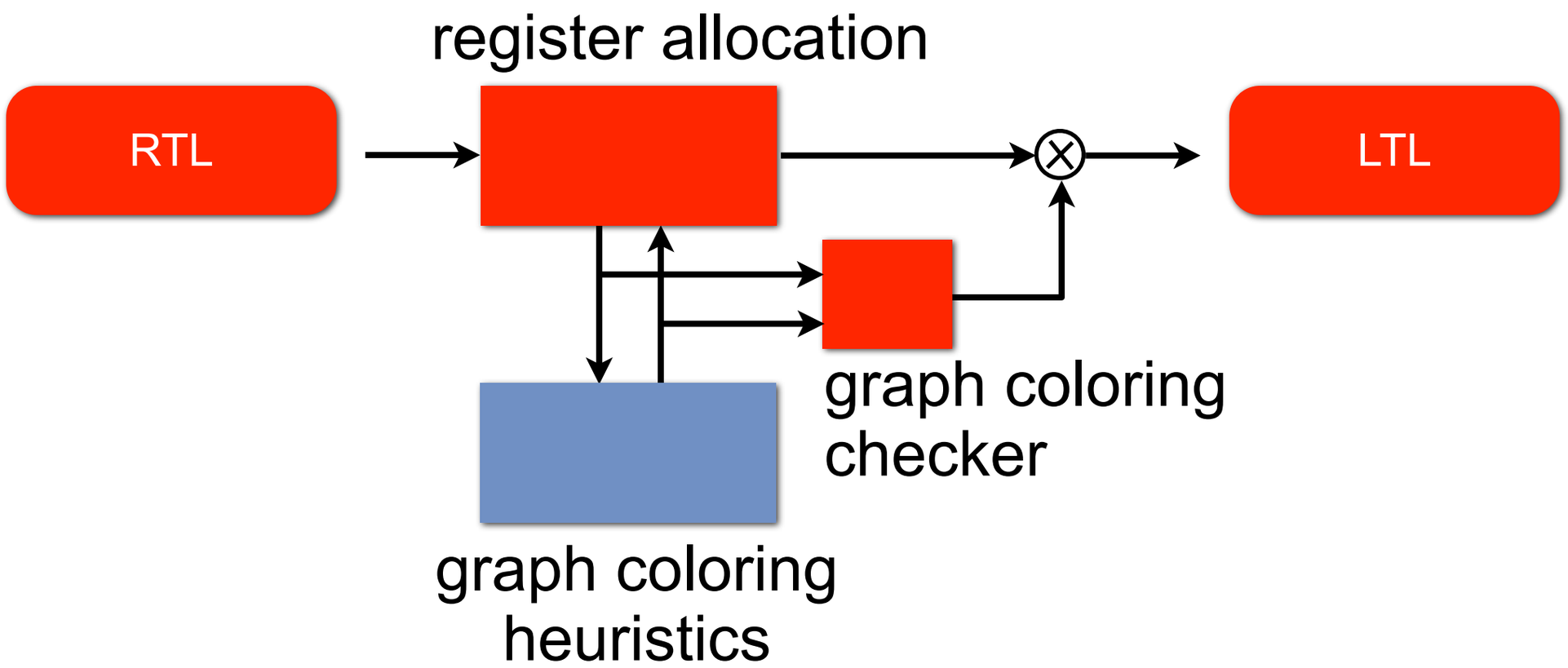


= formally verified

= not verified

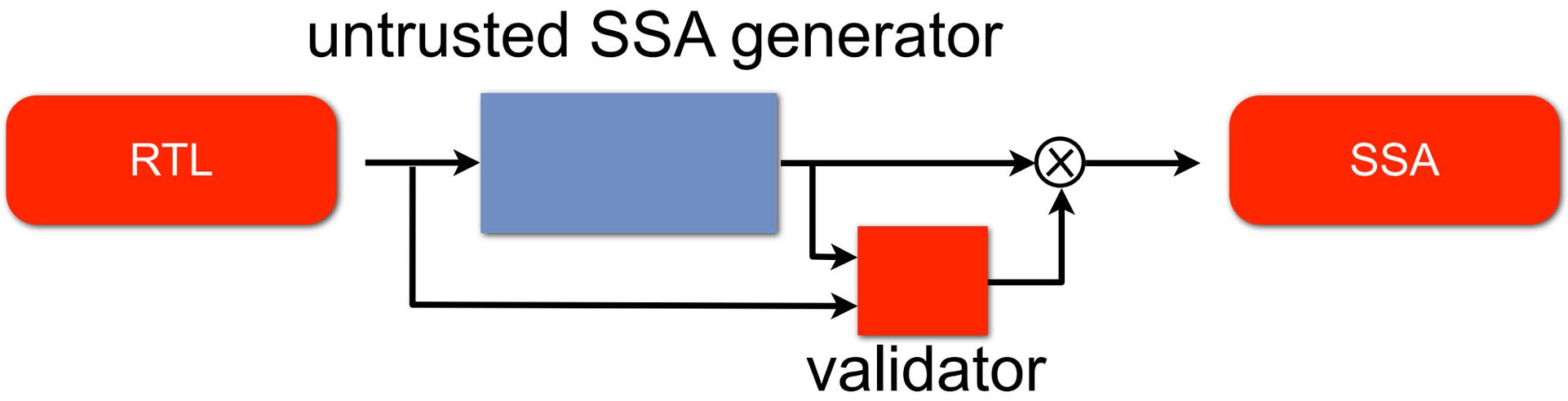
External solver with verified validation

Example: register allocation



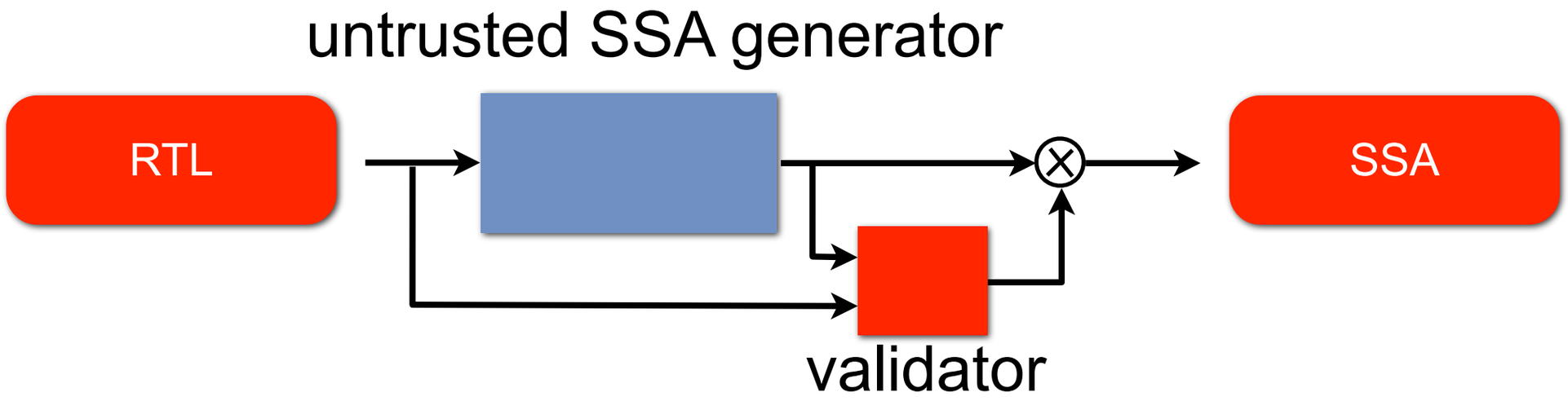
Verified translation validation

Example: SSA generation (in CompCert SSA extension)



Verified translation validation

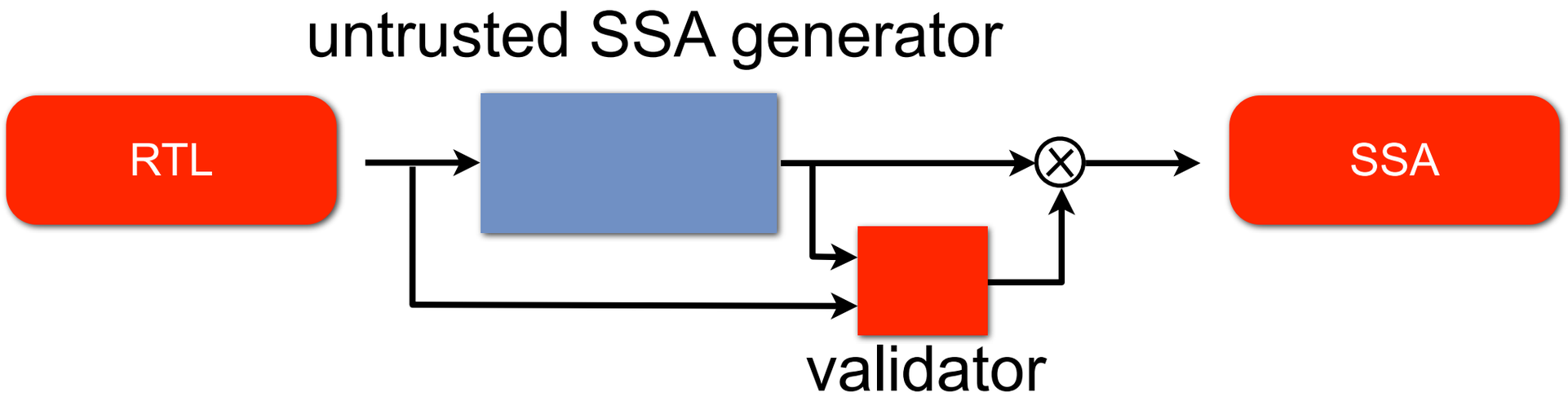
Example: SSA generation (in CompCert SSA extension)



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

Verified translation validation

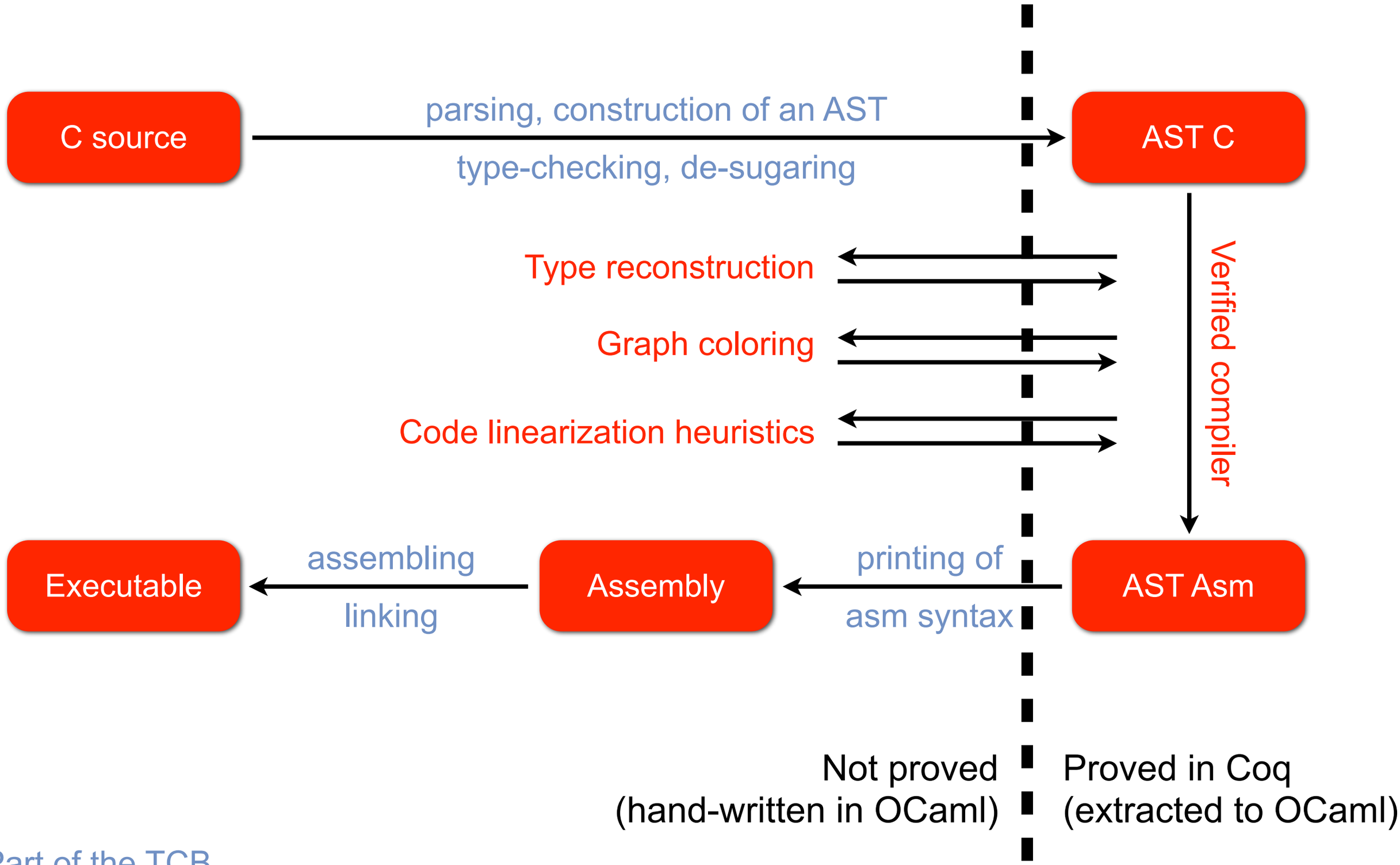
Example: SSA generation (in CompCert SSA extension)



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

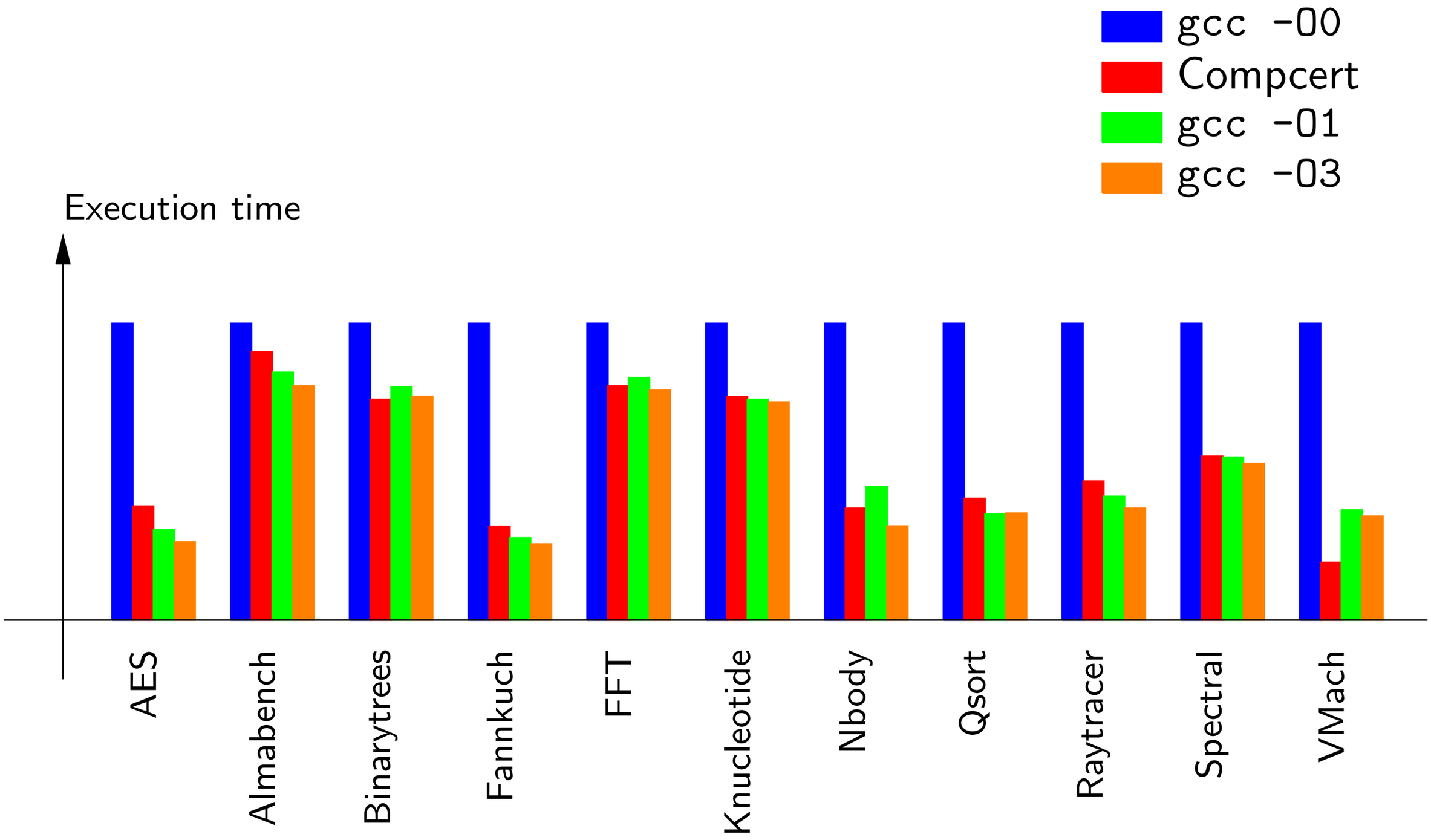


Part of the TCB

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)