Chamois: Formally verified compilation for optimisation and security

David Monniaux

CNRS / Verimag

June 26, 2026

Joint work with Sylvain Boulmé (associate prof), Basile Pesin (post-doc), Léo Gourdin, Cyril Six, Alexandre Bérard (PhD students), Benjamin Bonneau, Nicolas Nardino (interns)

Contents

CompCert

Formally verified C compiler, effort led by Xavier Leroy (INRIA then Collège de France) & Sandrine Blazy (U-Rennes / IRISA)

“If compilation succeeds, then the assembly program matches the C program.”

Formally verified: compiler written in Coq
+ correctness theorem proved in Coq, a proof assistant (mathematical proof, machine-checked)

Use

```
ccomp -Wall -c file1.c
ccomp -Wall -c file2.c
ccomp file1.c file2.c -o blah
```

Rationale for CompCert

Certain industries (avionics, nuclear...) must demonstrate that the object code is equivalent to the source.

Conventional approach

Disable optimizations
“Human” comparisons
“This compiler worked in other safety-critical projects”

CompCert

Use the mathematical proof

Versions under discussion

“Official” releases

(Xavier Leroy) https://github.com/AbsInt/CompCert

Commercial releases

Buy from AbsInt GmbH! https://www.absint.com/compcert/

Snapshots: https://github.com/AbsInt/CompCert-AbsInt-Releases

Our own “Chamois” branch

for agile development 

https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert

https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-Arsene

Targets

- x86 and x86-64 (not idiomatic)
- ARM ✓
- AArch64 ✓
- RISC-V 32- and 64-bit ✓
- PowerPC
- (Chamois only) Kalray KVX ✓
- (AbsInt commercial only) PeakTop
- (AbsInt commercial only) TriCore?

✓ = we've worked on it

Correctness theorem

execution = trace of “externally visible events” (**calls to external functions, volatile variables accesses**)

The trace at assembly matches the C trace
(more precisely: the assembly trace matches one of the acceptable behaviors of the C program, e.g. w.r.t. nondeterministic order of evaluation)

Mostly obtained by **forward simulation**: (assembly simulates C) through “match” relations

Example of lemma

```
Lemma type_of_operation_sound:
  forall op vl sp v m,
  is_special_op op = false ->
  eval_operation genv sp op vl m = Some v ->
  Val.has_type v (snd (type_of_operation op)).
```

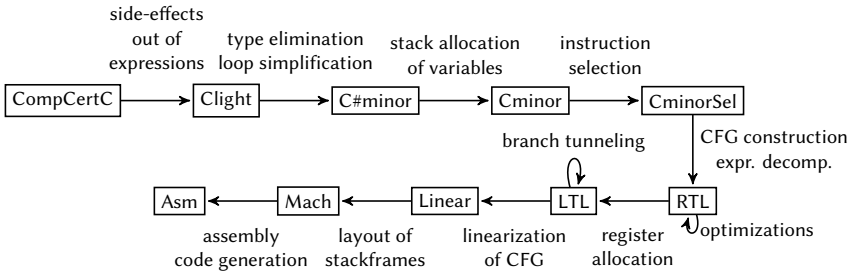
Forward simulations

Lockstep "One step of the program before transformation maps to one step after transformation."
$\sigma_1 \to_e \sigma_2$ and $m(\sigma_1, \sigma_1')$ then there exists σ_2' such that $\sigma_1' \to_e \sigma_2'$ and $m(\sigma_3, \sigma_2')$
e = "observable events"
e.g. "replace $x \times y$ by a move from a register already containing that expression"

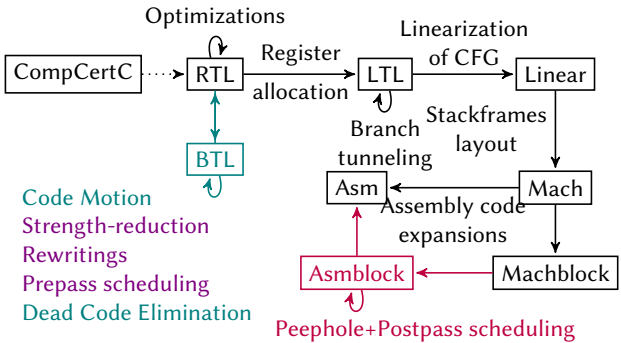
Plus "One step maps to several steps."
e.g. function call from one instruction to many (move operands to registers / stack etc.)

Star "Several steps map to several steps."

CompCert's intermediate languages



Backend phases



Legend:
Violet: AArch64+ARMv7+RISC-V+KVX
Red: AArch64+KVX (ARMv7 work in progress)
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)

The main difficulty

Optimizations etc. described quite informally in the literature.

Find good formalism definitions etc. to make proofs easier.

Same as with “formal mathematics” (Xena Project)

Contents

A menu

1. oysters
2. veal blanquette
   - 2.1 prepare blanquette
   - 2.2 cook it
3. millefeuille
   - 3.1 **puff pastry**
     - 3.1.1 fold 1, wait 30 minutes
     - 3.1.2 fold 2, wait 30 minutes
     - 3.1.3 fold 3, wait 30 minutes
     - 3.1.4 fold 4, wait 30 minutes
     - 3.1.5 fold 5
   - 3.2 cream

Scheduling

"Official" CompCert produces instructions roughly in the source ordering.

Not the best execution order in general!

Especially on in-order cores.

Our solution: **verified scheduling**

Example

$r_1 := a * b$
$r_3 := a - b$
$r_2 := r_1 + c$
branch($a > 0$, EXIT1)

$r_3 := a - b$
$r_4 := a * b$
$r_2 := r_4 + c$
branch($a > 0$, EXIT1)

r_1 and r_4 are both dead at EXIT1 and at final point.

These two blocks are **equivalent**: in both cases
$r_2 = (a * b) + c$ and $r_3 = a - b$

Acceptable refinement

$r_1 := a * b$
$r_3 := a - b$
$r_2 := r_1 + c$
$r_5 := a/b$
branch($a > 0$, EXIT1)

$r_3 := a - b$
$r_4 := a * b$
$r_3 := r_4 + c$
branch($a > 0$, EXIT1)
$r_5 := a/b$

r_5 dead on EXIT1.

On x86, the division may fail:

- it's allowed to move it beyond the branch
- the converse is not allowed

Peephole optimizer

Replace instructions by other sequences
esp. fused memory accesses (cannot be done earlier, because each load/store accesses one register, not several)

(AArch64)

```
str     x19, [sp, #16]
str     x20, [sp, #24]
movz    x20, #0, lsl #0
str     x21, [sp, #32]
adrp    x21, sl1
add     x21, x21, #:lo12:sl1
str     x22, [sp, #40]
```

```
stp     x19, x20, [sp, #16]
movz    x20, #0, lsl #0
stp     x21, x22, [sp, #32]
adrp    x21, sl1
mov     x22, x20
add     x21, x21, #:lo12:sl1
```

VLIW bundling

(Kalray KVX)

```
TIFFReadAndRealloc:
        addd    $r17 = $r12, 0
        addd    $r12 = $r12, -128
        sd      -128[$r12] = $r12
;;
        get     $r16 = $ra
;;
        sd      8[$r12] = $r16
;;
        sd      16[$r12] = $r18
        addd    $r18 = $r3, 0
;;
        sd      24[$r12] = $r19
        make    $r19 = sl2
;;
        sd      32[$r12] = $r20
;;
        sd      40[$r12] = $r21
        addd    $r21 = $r4, 0
;;
```

```
TIFFReadAndRealloc:
        addd    $r17 = $r12, 0
        addd    $r12 = $r12, -128
        sd      -128[$r12] = $r12
;;
        get     $r16 = $ra
;;
        sd      8[$r12] = $r16
;;
        sq      16[$r12] = $r18r19
        addd    $r18 = $r3, 0
        make    $r19 = sl2
;;
        so      32[$r12] = $r20r21r22r23
        addd    $r21 = $r4, 0
        make    $r22, 1048576
        make    $r23 = sl1
;;
```

Loop rotation

```
while(c) {
  body;
}
```

turned into

```
if (c) {
  do {
    body;
  } while (c);
}
```

Allows precomputing the condition inside the loop body, changes 2 branches per iteration into 1.

Loop peeling

```
while(c) {
  body;
}
```

turned into

```
if (c) {
  body
  do {
    body;
  } while (c);
}
```

Makes sure that operations always executed inside the peeled loop body are "available" for the next loop body.
(Together with global subexpression elimination, performs loop-invariant code motion.)

Code morphisms

On control-flow graphs
Each node in the transformed program corresponds to a node in the original

Lockstep simulation = “the operations are the same on each side”

- loop unrolling
- loop peeling
- loop rotation
- factoring (= regroup CFG nodes according to equivalence/congruence/bisimulation)

Redundant computation elimination

Goals

- Replaces computation by move if value available in same register on all incoming paths
- Replaces conditional branch by unconditional branch if condition statically known on all incoming paths

Example: redundant tests

```
#include <stdlib.h>
#define ABORT {abort(); while(1);}

inline void array_put(int n, double *t, int i, double v) {
  if (i < 0 || i >= n) ABORT else t[i] = v;
}

inline double array_get(int n, double *t, int i) {
  if (i < 0 || i >= n) ABORT else return t[i];
}

void mul_update(int n, double *t, int i, double x){
  array_put(n, t, i, array_get(n, t, i) * x);
}
```

Lazy code motion

Hoist loop-invariant code out of loops.

Proved by glue invariants + symbolic execution.

```
void mul42(double *t, int n) {
  for(int i=0; i<n; i++)
    t[i] *= 42;
}
```

Move the constant 42 load out of the loop.

Note: two ways of doing code hoisting

- loop peeling + global subexpression elimination
  - \+ can move loads and other trapping operations out
  - \- cannot deal with partial redundancies
- hoisting
  - \- cannot move loads and other trapping operations out
  - \+ can deal with partial redundancies

```
void toto(double *t, int n, _Bool f) {
  for(int i=0; i<n; i++) {
    t[i] *= f ? 42 : 37.1;
  }
}
```

Strength reduction

```
for(int i=0; i<n; i++) {
  r += t[i];
}
```

Naive compilation: `t[i]` means multiplication/shift, add, load.
Yet the address differs only by a constant offset across iterations!

Strength reduction

- Identify values that differ (add/subtract) by a constant across iteration.
- Rewrite multiplications…into addition by constant.
- Prove the transformation correct using glue invariants + symbolic execution + arithmetic rewrite rules.

Store motion

Hoist store operations out of loops.

Proved by glue invariants + symbolic execution.

Example: complex sum-product

```
typedef struct { double re, im; } complex;

inline void sum_complex(complex *s, const complex *a, const
    complex *b) {
  double re = a->re + b->re;
  double im = a->im + b->im;
  s->re = re;
  s->im = im;
}

inline void mul_complex(complex *s, const complex *a, const
    complex *b) {
  double re = a->re * b->re - a->im*b->im;
  double im = a->re * b->im + a->im*b->re;
  s->re = re;
```

Example: complex sum-product

```
void sumproduct_complex_array(complex *s, int n, complex *a,
    complex *b) {
  complex r = {0., 0.}, p;
  for(int i=0; i<n; i++) {
    mul_complex(&p, a+i, b+i);
    sum_complex(&r, &r, &p);
  }
  s->re = r.re;
  s->im = r.im;
}
```

Compiled complex sum-product main loop

(RISC-V)

```
.L102:
        fld     f29, 0(x12)
        fld     f12, 0(x13)
        fld     f14, 8(x12)
        fld     f11, 8(x13)
        fmul.d  f30, f29, f12
        fmul.d  f2, f14, f12
        fmul.d  f28, f14, f11
        fmul.d  f5, f29, f11
        addi    x14, x14, 1
        addi    x13, x13, 16
        addi    x12, x12, 16
        fsub.d  f3, f30, f28
        fadd.d  f0, f5, f2
        fadd.d  f4, f4, f3
        fadd.d  f1, f1, f0
        blt     x14, x5, .L102
```

Why not fused multiply-add?

Replacing $a \times b + c$ by $fma(a, b, c)$

Why not fused multiply-add?

Replacing $a \times b + c$ by $fma(a, b, c)$
changes **semantics** (results differ)

(Perhaps could be justifiable in some kind of symbolic algebraic semantics?)

Contents

Stack canaries

On functions with local arrays etc. put a canary (special data) at the end.
If buffer overrun (“stack smashing”), abort execution.
Blocks attempts at corrupting the return address.

```
void swap(int n, int *a, int *b) {
  int tmp[10];
  for(int i=0; i<n; i++) tmp[i]=a[i];
  for(int i=0; i<n; i++) a[i]=b[i];
  for(int i=0; i<n; i++) b[i]=tmp[i];
}

int main() {
  static int ka[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  static int kb[15] = {31,32,33,34,35,36,37,38,39,20,21,22,23,24,25};
  swap(15, ka, kb);
}
```

```
$ ./localcopy
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Stack canaries

- prove **correctness** (the security measure does not disrupt legal program executions)
- does not prove **adequacy** (the security measure blocks attacks)
- how to define an **attacker model** ?

Hardened copying procedure

(RISC-V)

```
swap:
  mv  x30, x2
  addi  x2, x2, -64
  sd  x30, 0(x2)
  sd  x1, 8(x2)
  la  x14, __guard
  ld  x14, 0(x14)
  sd  x14, 56(x2)
  addi  x15, x0, 0
  bge  x0, x10, .L100
.L101:
  addi  x14, x2, 16
  slli  x13, x15, 2
  add  x14, x14, x13
  add  x13, x11, x13
  lw  x13, 0(x13)
  sw  x13, 0(x14)
  addi  x15, x15, 1
  mv  x13, x10
  blt  x15, x13, .L101
.L100:
  addi  x14, x0, 0
  bge  x0, x10, .L102
.L103:
  slli  x15, x14, 2
  add  x5, x11, x15
  add  x15, x12, x15
  lw  x13, 0(x15)
  sw  x13, 0(x5)
  addi  x14, x14, 1
  mv  x13, x10
  blt  x14, x13, .L103
.L102:
  addi  x14, x0, 0
  bge  x0, x10, .L104
.L105:
  slli  x13, x14, 2
  add  x11, x12, x13
  addi  x15, x2, 16
  add  x13, x15, x13
  lw  x15, 0(x13)
  sw  x15, 0(x11)
  addi  x14, x14, 1
  mv  x11, x10
  blt  x14, x11, .L105
.L104:
  ld  x12, 56(x2)
  la  x11, __guard
  ld  x11, 0(x11)
  beq  x12, x11, .L106
.L107:
  li  x10,21
  call  exit@plt
  j  .L107
.L106:
  ld  x1, 8(x2)
  addi  x2, x2, 64
  jr  x1
```

Branch target authentication

(AArch64, x86) (needs library / OS support)
Only allow branches to targets marked as legal branch targets

Implemented as changes to assembly emitter, no attempt at semantic characterization

Branch target authentication for x86

```
big_switch:
        endbr64
        subq    $8, %rsp
        leaq    16(%rsp), %rax
        movq    %rax, 0(%rsp)
        cmpl    $16, %edi
        jb      .L100
        movl    $99, %eax
        jmp     .L101
.L100:
        leaq    .L102(%rip), %rax
        movslq  (%rax, %rdi, 4), %rdx
        addq    %rdx, %rax
        notrack jmp     *%rax
```

Branch target authentication for AArch64

```
big_switch:
  hint    34 // bti c
  mov  x15, sp
  sub  sp, sp, #16
  str  x15, [sp, #0]
  str  x30, [sp, #8]
  cmp  w0, #16
  b.lo  .L100
  movz  w0, #99, lsl #0
  b  .L101
.L100:
  adr  x16, .L102
  add  x16, x16, w0, uxtw #3
  br  x16
.L102:  hint 36 // bti j
  b  .L103
  hint 36 // bti j
  b  .L104
```

Pointer authentication

(AArch64)
Return addresses are stored with additional tags (secret key in processor)
If overwritten with incorrectly tagged value, crash when return
Proof of **correctness** = instead of storing return address, store tagged return address

Possible extension: authentication on all pointers
Needs tagging at source level (type system)
Needs suitable semantic definitions for memory injections etc.

Branch target authentication for AArch64

```
big_switch:
  hint  25 // paciasp
  mov  x15, sp
  sub  sp, sp, #16
        ...
  ldr  x30, [sp, #8]
  add  sp, sp, #16
  hint  29 // autiasp
  ret  x30
```

Protection against faults

Laser / electromagnetic pulses that alter operations

Duplicate computations, add extra tracking variable, check consistency
(Only in Chamois-Arsene)

Show

- correctness (trace preservation)
- **adequacy** against a **nonstandard semantics** with faults

Vulnerable tests

```
_Bool verify(int n, const char *a, const char *b) {
  for(int i=0; i<n; i++) {
    if (a[i] != b[i]) return 0;
  }
  return 1;
}
```

Vulnerable tests

(RISC-V)

```
verify:
  .cfi_startproc
  mv       x30, x2
  addi     x2, x2, -16
  .cfi_adjust_cfa_offset  16
  sd       x30, 0(x2)
  sd       x1, 8(x2)
  .cfi_rel_offset x1, 8
  addi     x14, x0, 0
  bge      x0, x10, .L100
.L101:
  add      x13, x11, x14
  lbu      x15, 0(x13)
  add      x13, x12, x14
  lbu      x13, 0(x13)
  beq      x15, x13, .L102
  addiw    x10, x0, 0
  j        .L103
.L102:
  addi     x14, x14, 1
  mv       x13, x10
  blt      x14, x13, .L101
.L100:
  addiw    x10, x0, 1
.L103:
  addi     x2, x2, 16
  jr       x1
```

Control-flow integrity hardening

```
__attribute((harden(”control_flow_checking”))) _Bool verify(
    int n, const char *a, const char *b) {
  for(int i=0; i<n; i++) {
    if (a[i] != b[i]) return 0;
  }
  return 1;
}
```

Hardened program

(RISC-V)

```
verify:
  mv  x30, x2
  addi  x2, x2, -16
  sd  x30, 0(x2)
  sd  x1, 8(x2)
  addiw  x14, x0, 0
  addiw  x13, x0, 13
  bge  x0, x10, .L100
.L101:
  addiw  x15, x0, 11
.L102:
  blt  x14, x10, .L103
  xor  x10, x13, x15
  addiw  x11, x0, 2
  bne  x10, x11, .L104
  addiw  x10, x0, 1
  j  .L105
.L103:
  xor  x15, x13, x15
  addiw  x13, x0, 6
  bne  x15, x13, .L104
  mv  x13, x14
  add  x5, x11, x13
  lbu  x5, 0(x5)
  add  x13, x12, x13
  lbu  x13, 0(x13)
  beq  x5, x13, .L106
  addiw  x6, x0, 3
  j  .L107
.L106:
  addiw  x6, x0, 5
.L107:
  beq  x5, x13, .L108
  xor  x12, x15, x6
  addiw  x14, x0, 5
  bne  x12, x14, .L104
  addiw  x10, x0, 0
.L105:
  ld  x1, 8(x2)
  addi  x2, x2, 16
  jr  x1
.L108:
  xor  x13, x15, x6
  addiw  x15, x0, 3
  bne  x13, x15, .L104
  addiw  x14, x14, 1
  xori  x13, x13, 14
  blt  x14, x10, .L101
.L100:
  addiw  x15, x0, 15
  j  .L102
.L104:
  li  x10,21
  call  exit@plt
  j  .L104
```

Ideas à la SecSwift

- **duplicate tests** (4 cases, 2 absurd)
- have a **special variable** updated through xor
- check it periodically for correct value
- if one manages to jump in the middle of a block: incorrect value

Interprocedural hardening

```
_Bool f1(int a, int b) {
  return a < b;
}

_Bool f2(int a, int b, int c) {
  return f1(a, b) && f2(b, c);
}
```

Interprocedural hardening

For each function `f` emit both `f` and `ipcfc..f`

- `f`: called using normal calling convention, sets up appropriate parameters, calls `ipcfc..f`
- `ipcfc..f`: takes an extra parameter, initial value of the xor-flow

Criticism on attack surface?

Interprocedural hardening

For each function `f` emit both `f` and `ipcfc..f`

- `f`: called using normal calling convention, sets up appropriate parameters, calls `ipcfc..f`
- `ipcfc..f`: takes an extra parameter, initial value of the xor-flow

Criticism on attack surface? `f` extra attack surface — correctly initialize variable
Remove useless unprotected functions: make them `static` or use `-ffunction-sections -Wl,-gc-sections` or equivalent to remove never called functions

Caveats for proof of adequacy

Proof of adequacy on high-level IR
Single fault only

Could be destroyed by subsequent passes (optimization identifies redundant tests)
(We did validate against **Lazart**, an attacking symbolic execution tool)

Precaution against removal of countermeasure by optimization: **opaque copies**
(also available as builtins)

Contents

Advanced scheduling

Scheduling = compute “optimal” order

Objective function: **makespan**

Keep maximal number of registers below number of hardware registers

Combinatorial optimization
Parametric complexity

Memcpy

Current CompCert: builtin memcpy is expanded by trusted, unproved OCaml code (TCB)
Bugs found in the expansion code [ESOP 2022]
Reduce the trusted computing base!

Replace unproved expansion code by (almost completely) formally verified code (reduce TCB)

Merge consecutive copy operations (for efficiency + code size on some programs generated from data-flow languages)

How to prove things

- Need to rework analyses.
- Think carefully about invariants, what needs to be proved, what needn't.
- Possibly split complex optimizations into distinct phases with simple specification.
- Possibly split thing into an oracle and a checker.
- Any unclear / badly designed aspect of semantics will bite you.

Cheri

Verified compilation for CHERI Risc-V or AArch64?

CHERI = fat pointers with **capabilities**

Rust

Verified compilation of Rust?

- thesis in progress on verified borrow-checker
- submitted an ANR project (INRIA Paris / Saclay / LIP6 / Grenoble)
- start a larger project? (suggestions of ERC Synergy, **looking for partners**)

Semantics

Current situation

huge proofs talking about **global memory invariants** and **every level of the call stack**

Time modularity

move **away from small-step semantics** (be able to “jump through” function calls...)

Space modularity

be able to reason about **memory separation** efficiently

Questions?