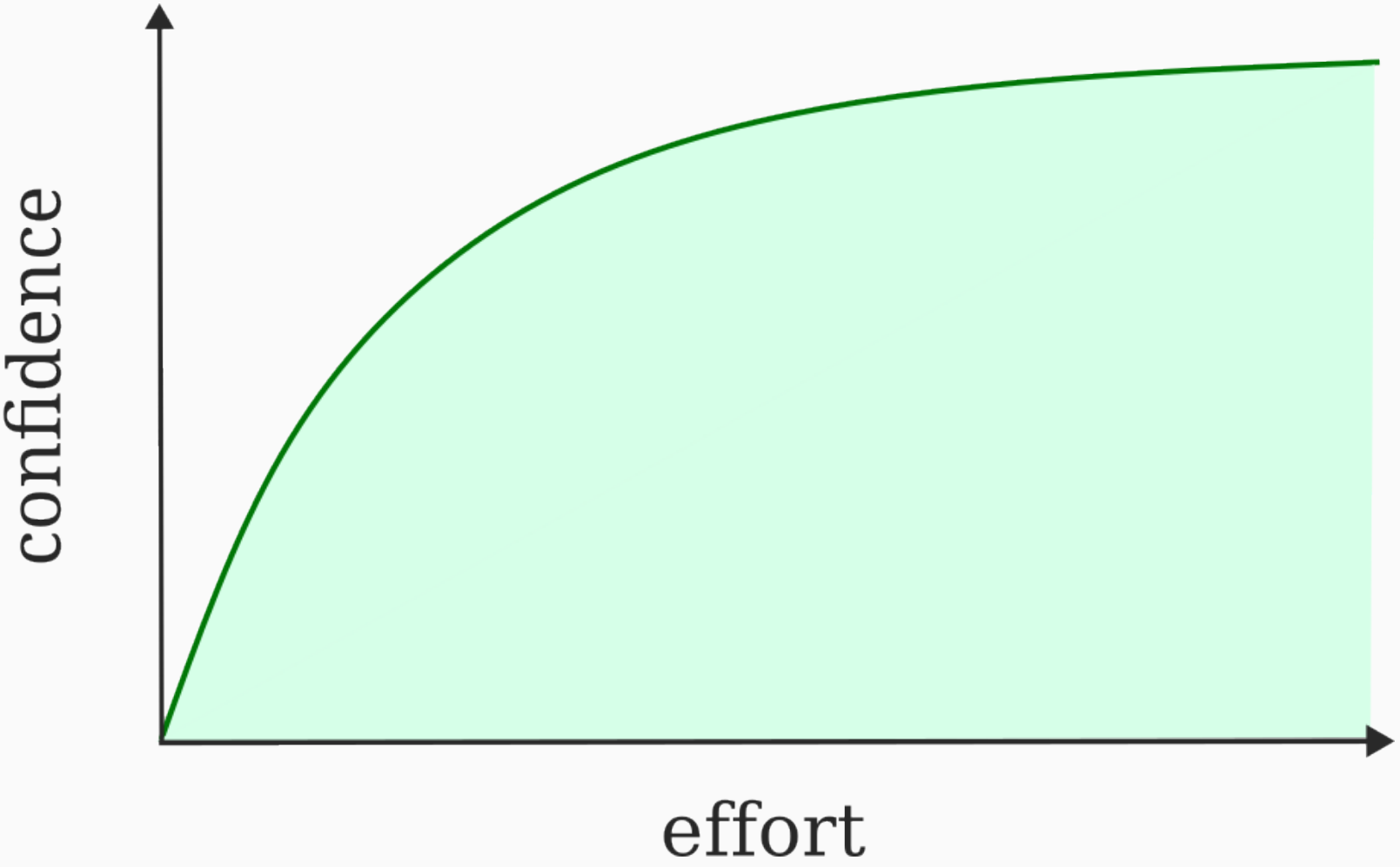
How to Minimize Bugs in Cryptography Code

Jade Philipoom

2025-12-28

Everyone makes mistakes


confidence
effort

```
encrypt(key, message)
```

```
encrypt(key, message)
```

```
encrypt(0x123456..., "test message") = 0xf93274b...
```

```
encrypt(key, message)
```

```
encrypt(0x123456..., "test message") = 0xf93274b...
```

```
k = random()
m = random()
decrypt(k, encrypt(k, m)) = m
```

```
encrypt(key, message)
```

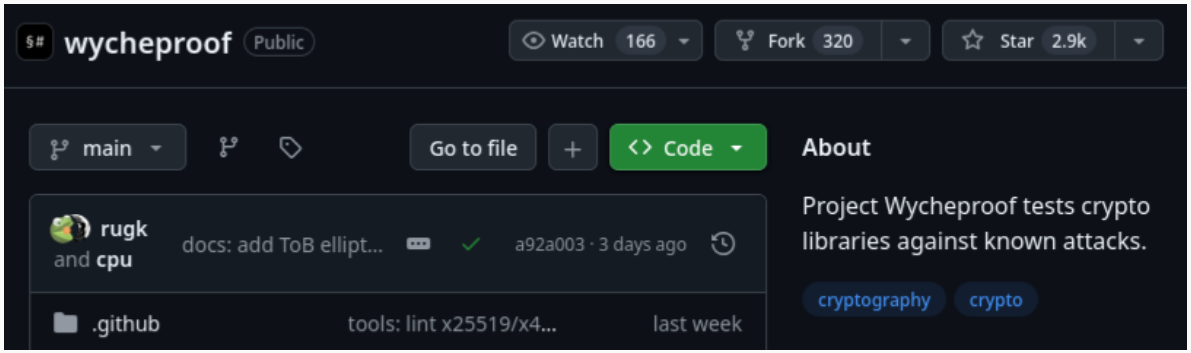
```
encrypt(0x123456..., "test message") = 0xf93274b...
```

```
k = random()
m = random()
decrypt(k, encrypt(k, m)) = m
```

```
k = random()
m = random()
encrypt(k, m) = other_lib_encrypt(k, m)
```

Clever testing

Wycheproof: the first line of defense




```
"tests": [
  {
    "tcId": 121,
    "comment": "regression test for arithmetic error",
    "flags": [
      "TinkOverflow"
    ],
    "msg": "26d5f0631f49106db58c4cfc903691134811b33c",
    "sig": "9588e02bc815649d359ce710cdc69814556dd8c8ba
    "result": "valid"
  }
```

Coverage-guided fuzzing tries to generate inputs that cover all lines in your code.

```
if (x >> 32 == 0x39c3) {
    // happens only 1/2^32 random tests
    return a;
} else {
    ...
    return b;
}
```

Same behavior as the previous code, but now it's harder for coverage-guided fuzzing to realize if the `a` case is not tested.

```
uint64_t y = (x >> 32) ^ 0x39c3;
uint64_t c = (0 - y) >> 63; // 0 if y=0, otherwise 1
uint64_t bmask = 0 - c; // all 1s if c=1, 0 if c=0
uint64_t amask = ~bmask; // all 1s if c=0, 0 if c=1
...
return (a & amask) | (b & bmask); // either a or b
```

Static analysis

Type systems are a form of static analysis!

```
void foo(int a) {
    ...
}

char bar[] = "bar";
foo(bar); // compiler error!
```

```
void foo(nonsecret_t a) {
    ...
}

secret_t key = get_key();
foo(key); // compiler error!
```

In C, static analysis tools can also check for buffer/integer overflows and undefined behavior.

```
void foo(int *a) {
    a[10] = 0;
    ...
}

int bar[3] = {1, 2, 3};
foo(bar);
```

The Rust borrow checker is also an example of static analysis.

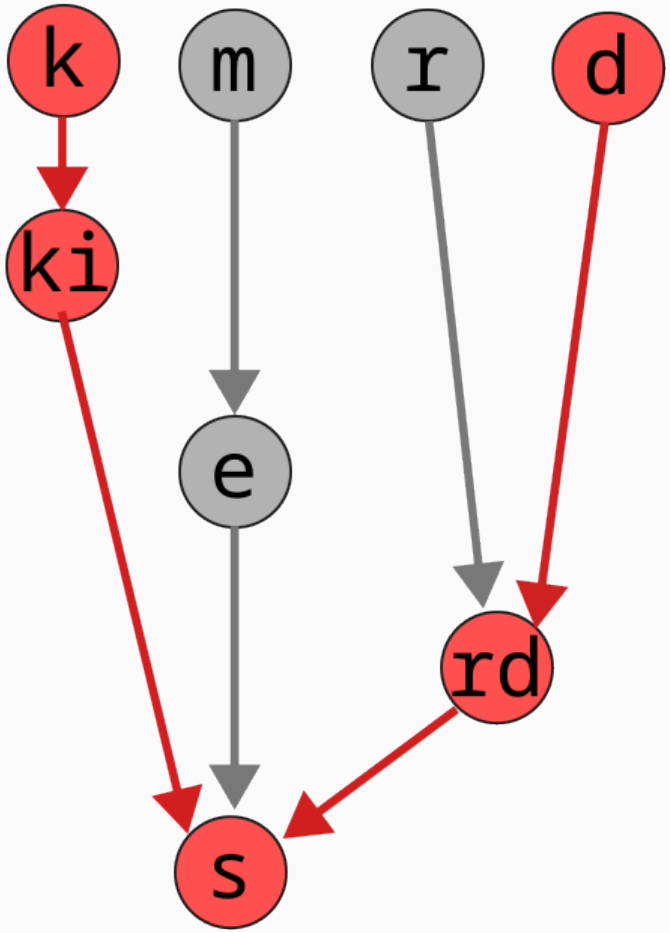
```
fn foo(a: &mut [i32]) {
    for _ in a.iter() {
        a[0] += 1;
    }
}
```

```
error[E0506]: cannot assign to `a[_]` because it is borrowed
 --> src/main.rs:4:9
  |
3 |     for _ in a.iter() {
  |              --------
  |              |
  |              `a[_]` is borrowed here
  |              borrow later used here
4 |         a[0] += 1;
  |         ^^^^^^^^^ `a[_]` is assigned to here but it was already borrowed
```

```python
def check_loop(program: OTBNProgram) -> CheckResult:
    '''Check that loops are properly formed.

    Performs three checks to rule out certain classes of loop errors and
    undefined behavior:
    1. For loopi instructions, check that the number of iterations is > 0.
    2. Ensure that loops do not end in control-flow instructions such as jal or
       bne, which will raise LOOP errors.
    3. Checks that there is no branching into or out of loop bodies.
    4. For nested loops, the inner loop is completely contained within the
       outer loop.
    '''
    loops = _get_loops(program)
    out = CheckResult()
    out += _check_loop_iterations(program, loops)
    out += _check_loop_end_insns(program, loops)
    out += _check_loop_stack(program, loops)
    out += _check_loop_inclusion(program, loops)
    out.set_prefix('check_loop: ')
    return out
```

```
$ check_loop.py bad_loop.elf
check_loop: ERROR: Control flow instruction (jal) at
end of loop at PC 0x14
```




```
ki = pow(k,-1)

e = hash(m)

rd = r * d

s = ki * (e + rd)
```

```
$ check_const_time.py --verbose x25519.elf\
    --subroutine X25519 --secrets w8
Analyzing routine X25519 with initial secrets ['w8']
PASS
```

Formal methods

SAT and SMT solvers

C Bounded Model Checker (CBMC)

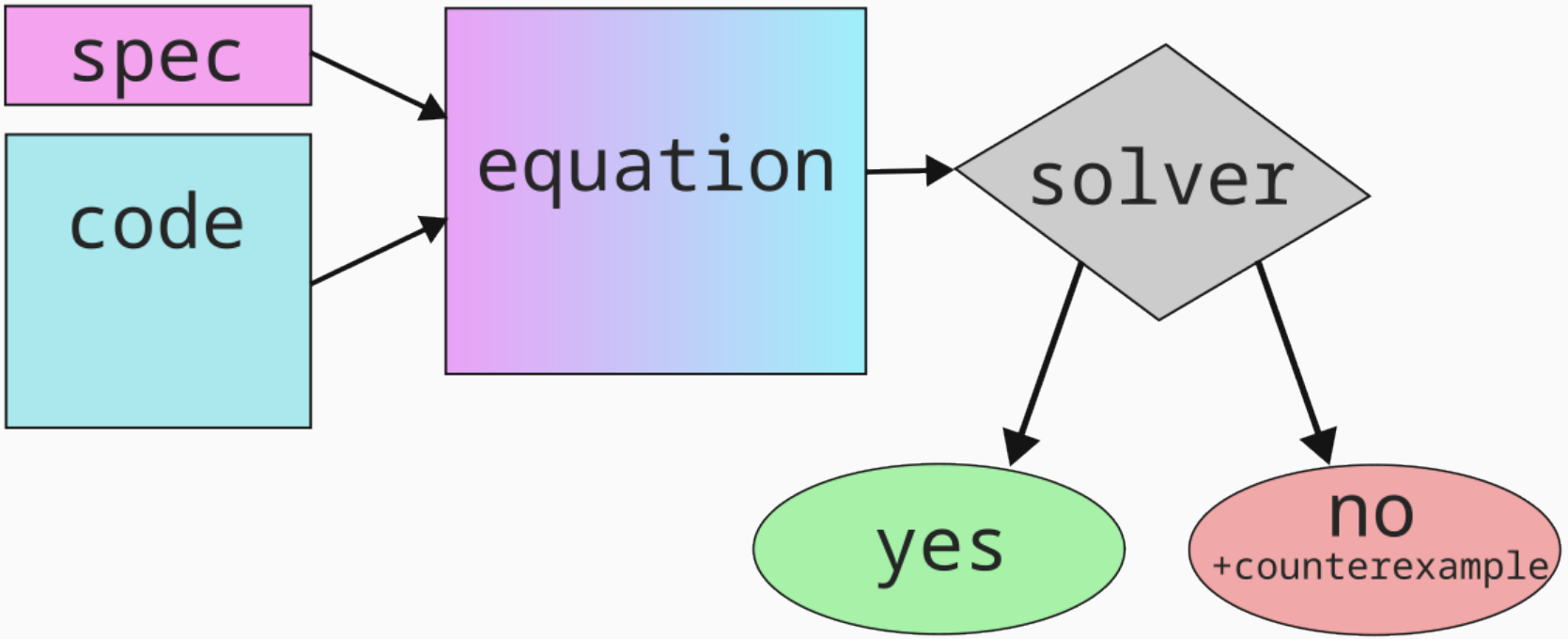
```
MLK_STATIC_TESTABLE void mlk_poly_ntt_c(mlk_poly *p)
__contract__(
  requires(memory_no_alias(p, sizeof(mlk_poly)))
  requires(array_abs_bound(p->coeffs, 0, MLKEM_N, MLKEM_Q))
  assigns(memory_slice(p, sizeof(mlk_poly)))
  ensures(array_abs_bound(p->coeffs, 0, MLKEM_N, MLK_NTT_BOUND))
)
{
```

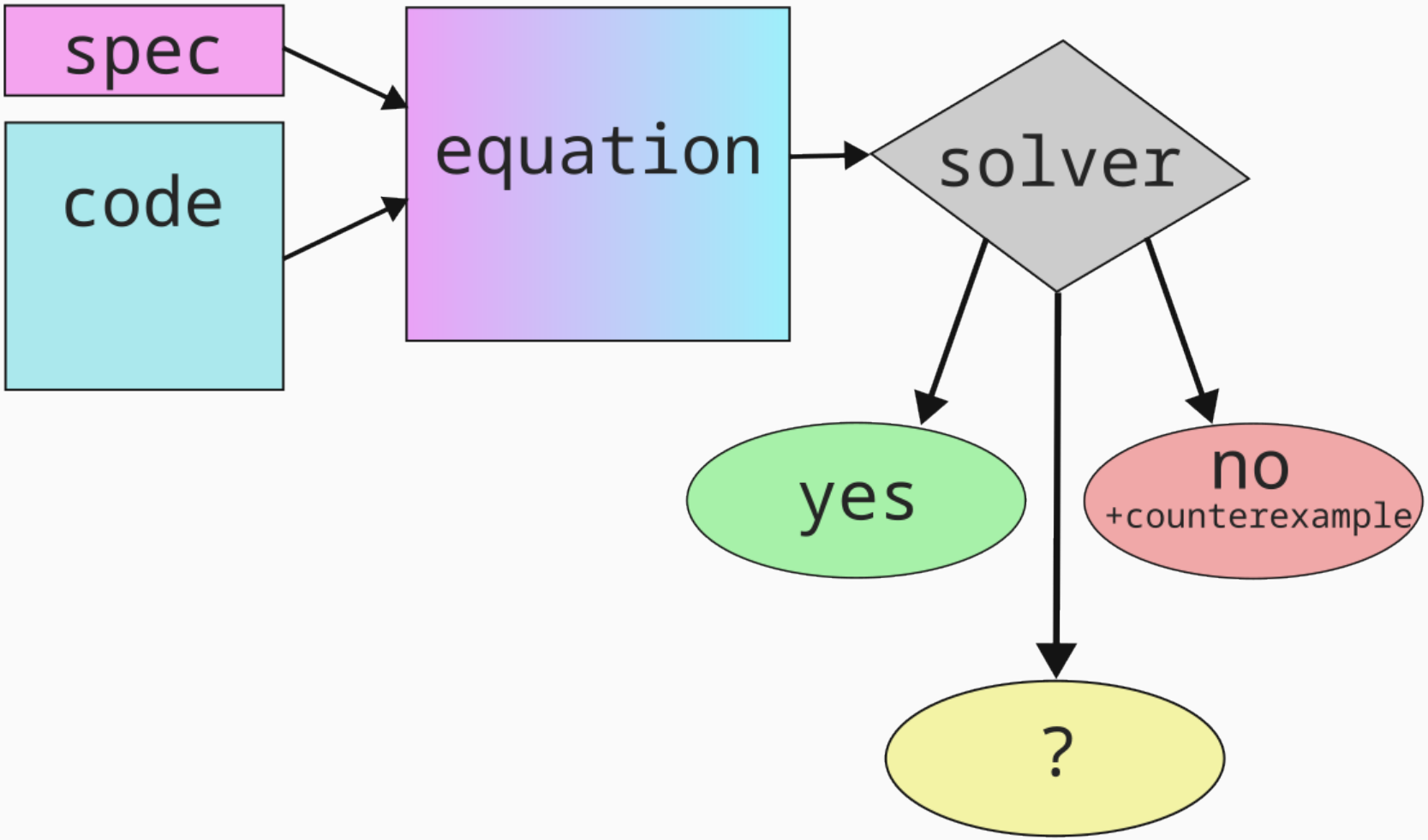
This is a real example from the mlkem-native project, and ensures that during a particular polynomial transformation (NTT) the upper bounds on each coefficient in the polynomial stay low enough.

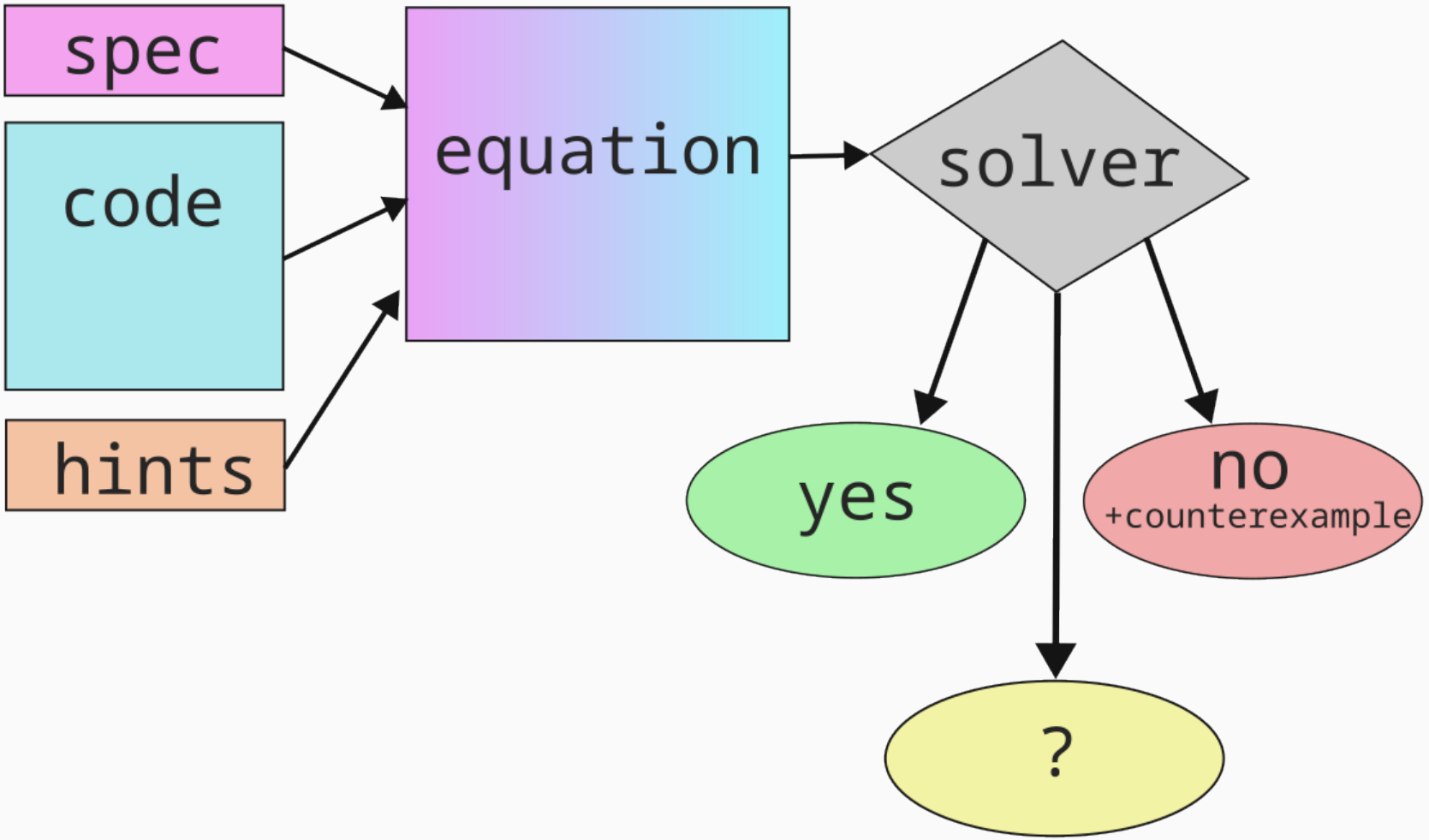
Proof-aware programming languages (Dafny, F*)

```
method Maximum(values: seq<int>) returns (max: int)
  requires values != []
  ensures max in values
  ensures forall i | 0 <= i < |values| :: values[i] <= max
{
  max := values[0];
  for idx := 0 to |values|
    invariant max in values
    invariant forall j | 0 <= j < idx :: values[j] <= max
  {
    if max < values[idx] {
      max := values[idx];
    }
  }
}
```

Example from Dafny's Github repository: full correctness for a small program. Dafny and Z3 solve this fully automatically.


spec
code
equation
solver
yes
no
+counterexample

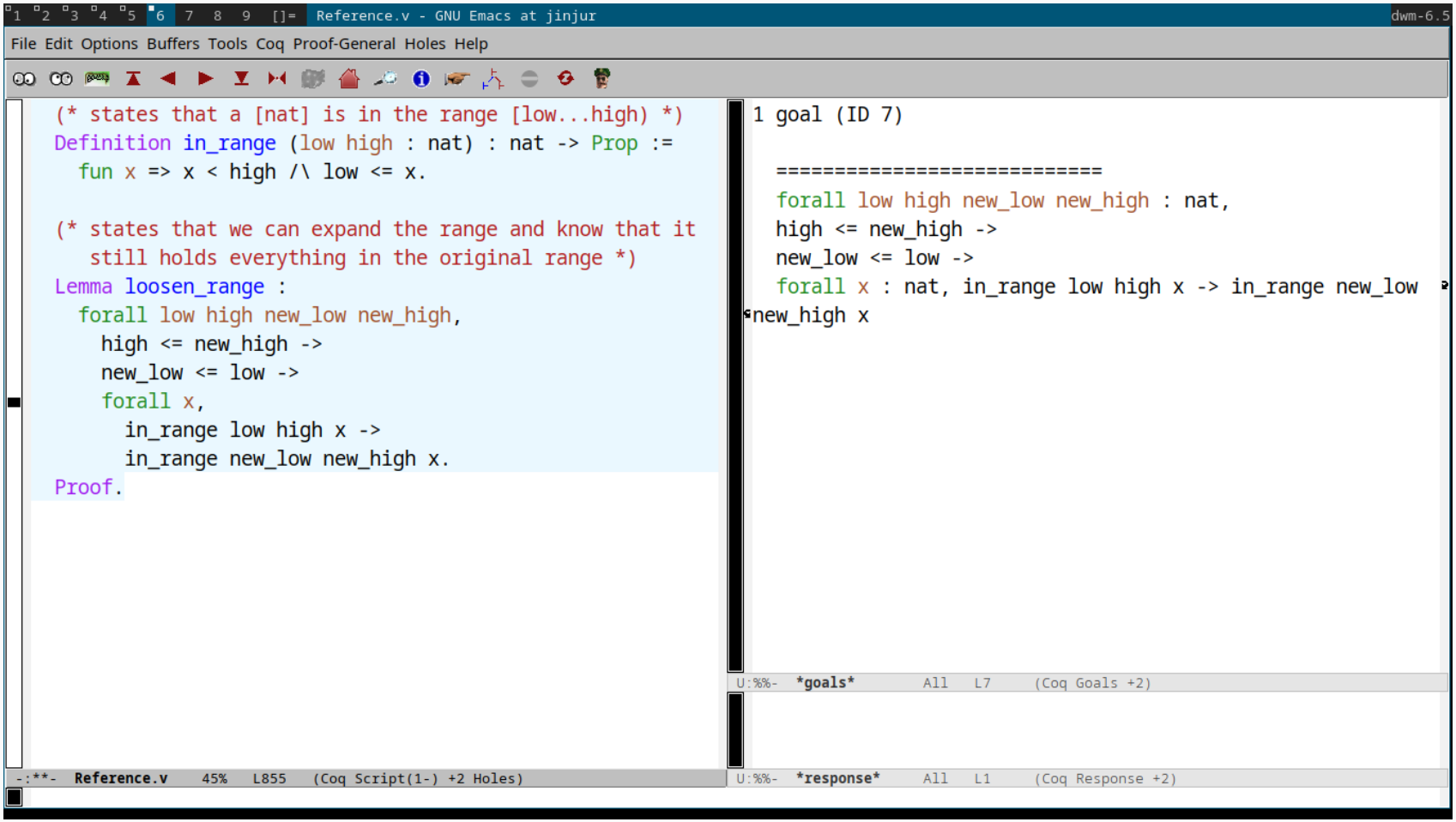

spec
code
equation
solver
yes
no
+counterexample
?

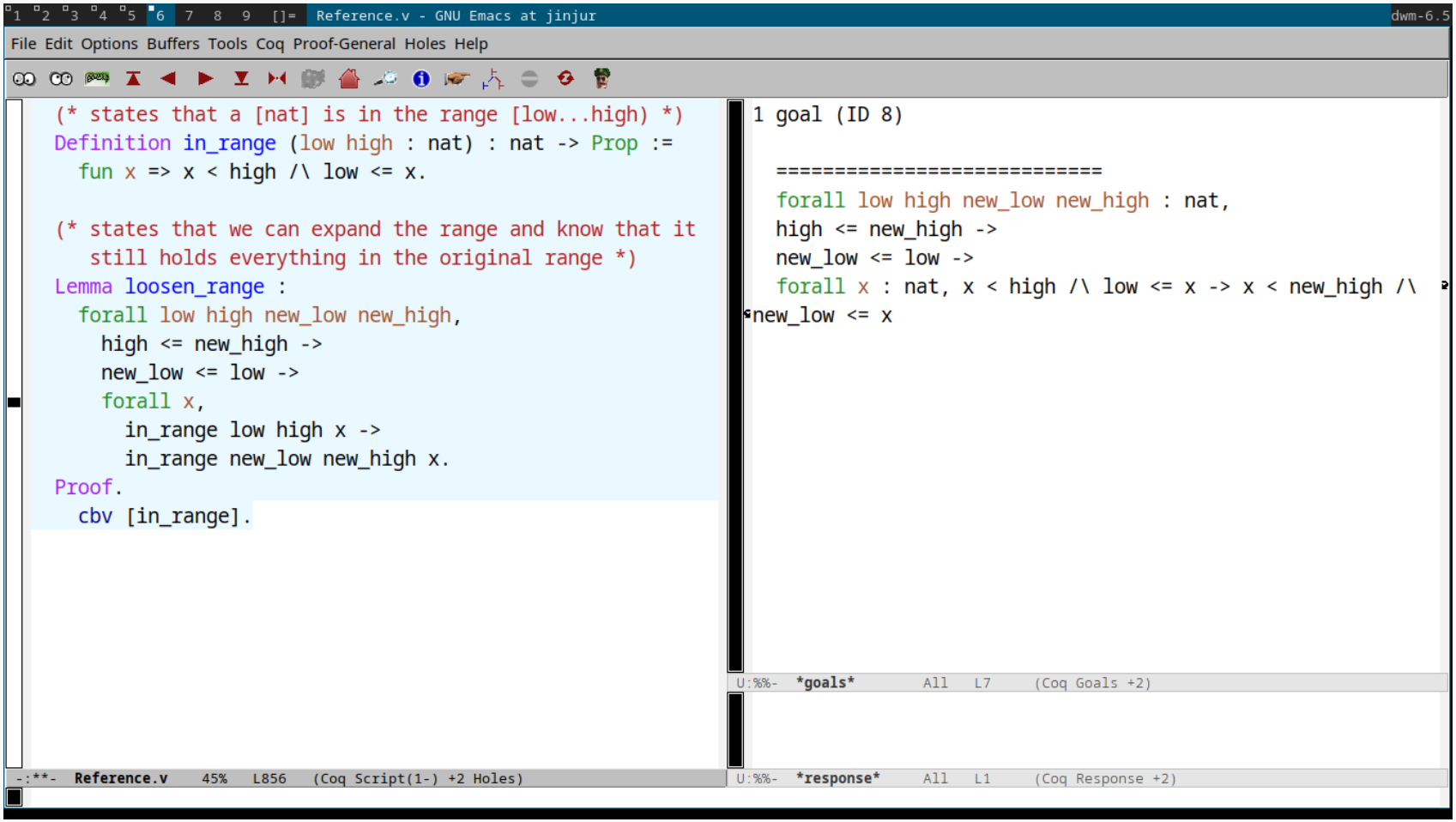

spec
code
hints
equation
solver
yes
no
+counterexample
?

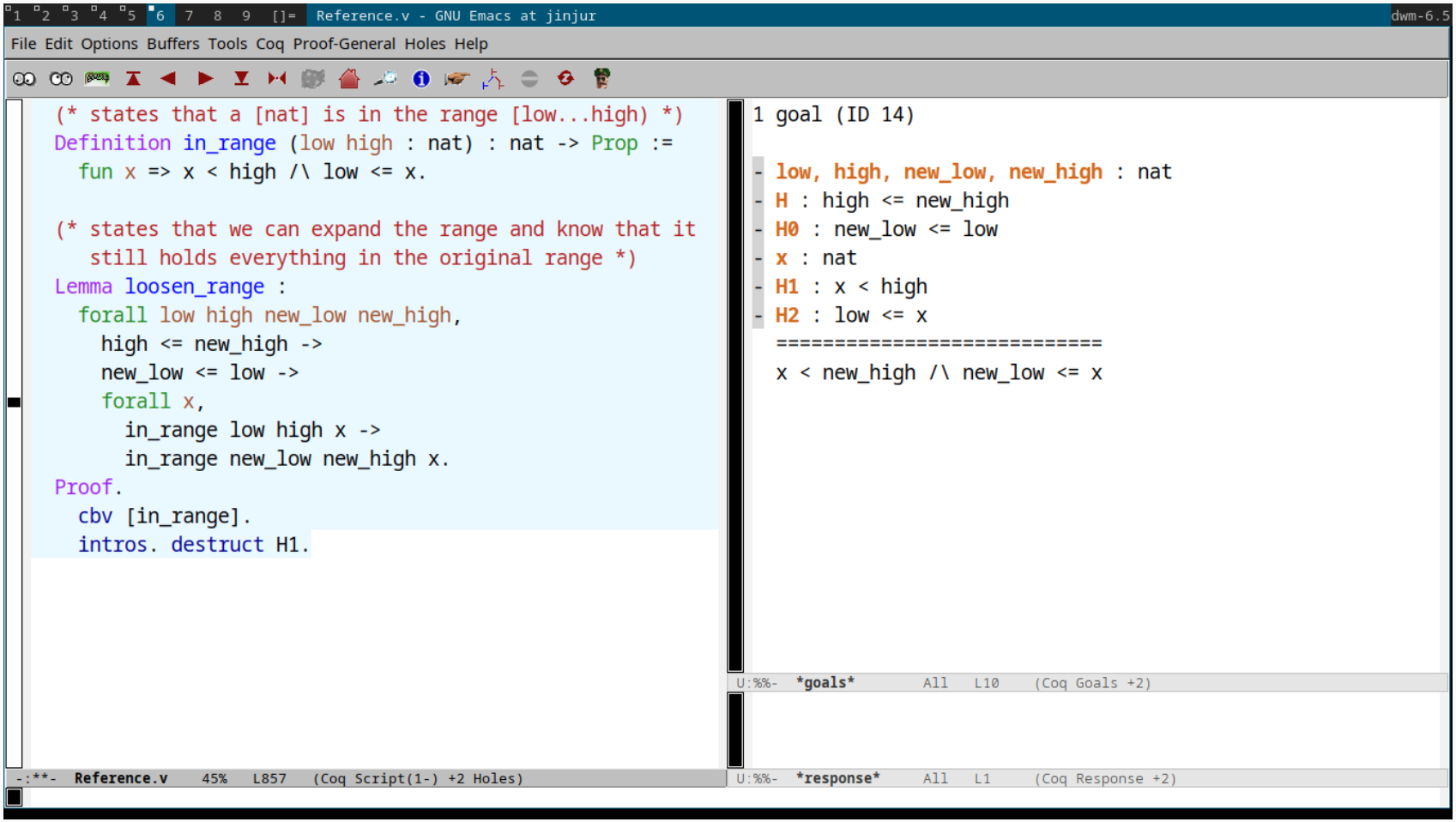
Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software

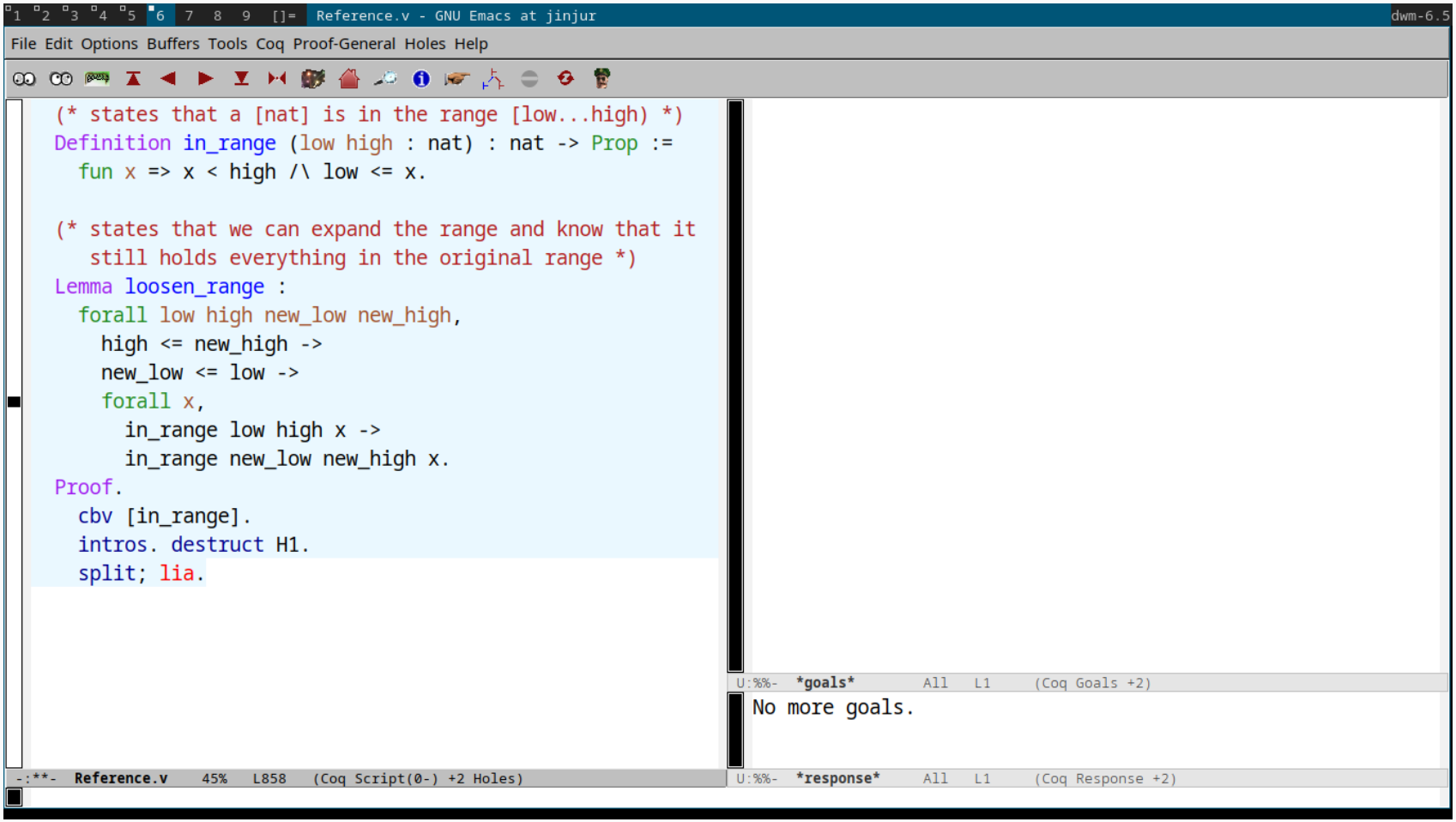
"Our experience with SMT solving was, on the whole, positive—it is hard to imagine proofs at the scale of ours being done without heavy automation. That said, we also confronted several challenges, including the opacity of SMT solvers and occasional sensitivity to small changes in verification conditions."

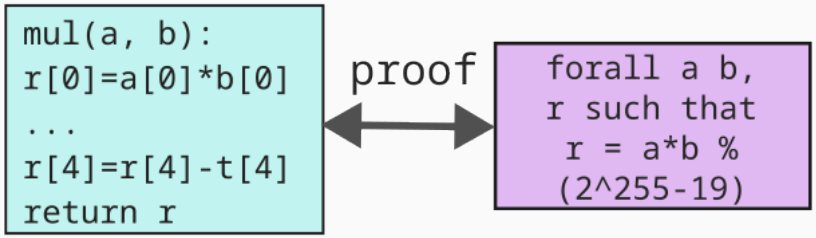
Proof Assistants

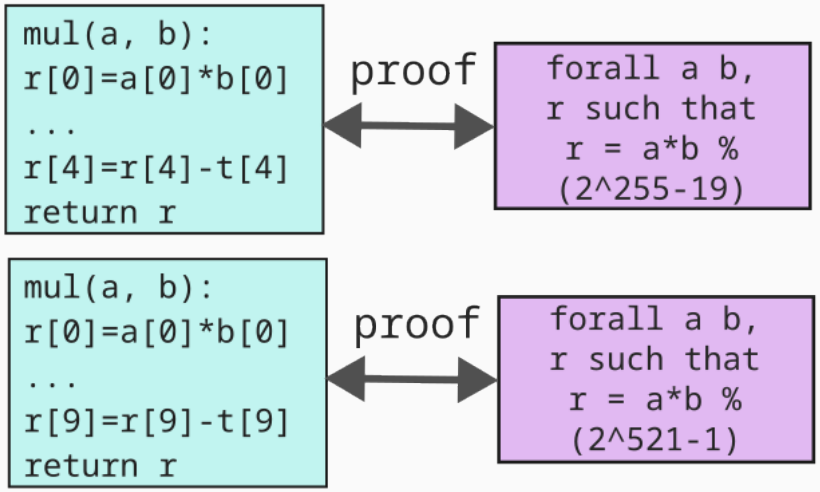

1 2 3 4 5 6 7 8 9 []= Reference.v - GNU Emacs at jinjur dwm-6.5
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* states that a [nat] is in the range [low...high) *)
Definition in_range (low high : nat) : nat -> Prop :=
fun x => x < high /\ low <= x.
(* states that we can expand the range and know that it
still holds everything in the original range *)
Lemma loosen_range :
forall low high new_low new_high,
high <= new_high ->
new_low <= low ->
forall x,
in_range low high x ->
in_range new_low new_high x.
Proof.
1 goal (ID 7)
============================
forall low high new_low new_high : nat,
high <= new_high ->
new_low <= low ->
forall x : nat, in_range low high x -> in_range new_low new_high x
U:%%- *goals* All L7 (Coq Goals +2)
-:**- Reference.v 45% L855 (Coq Script(1-) +2 Holes)
U:%%- *response* All L1 (Coq Response +2)

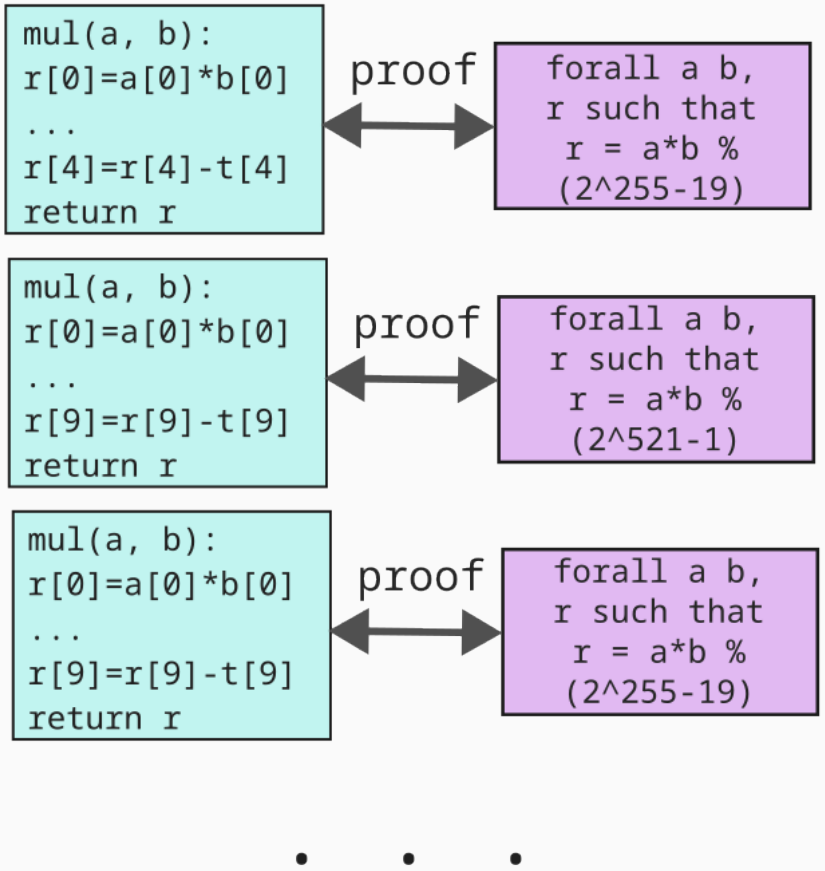

1 2 3 4 5 6 7 8 9 []= Reference.v - GNU Emacs at jinjur dwm-6.5
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* states that a [nat] is in the range [low...high) *)
Definition in_range (low high : nat) : nat -> Prop :=
fun x => x < high /\ low <= x.
(* states that we can expand the range and know that it
still holds everything in the original range *)
Lemma loosen_range :
forall low high new_low new_high,
high <= new_high ->
new_low <= low ->
forall x,
in_range low high x ->
in_range new_low new_high x.
Proof.
cbv [in_range].
1 goal (ID 8)
============================
forall low high new_low new_high : nat,
high <= new_high ->
new_low <= low ->
forall x : nat, x < high /\ low <= x -> x < new_high /\
new_low <= x
U:%%- *goals* All L7 (Coq Goals +2)
-:**- Reference.v 45% L856 (Coq Script(1-) +2 Holes)
U:%%- *response* All L1 (Coq Response +2)

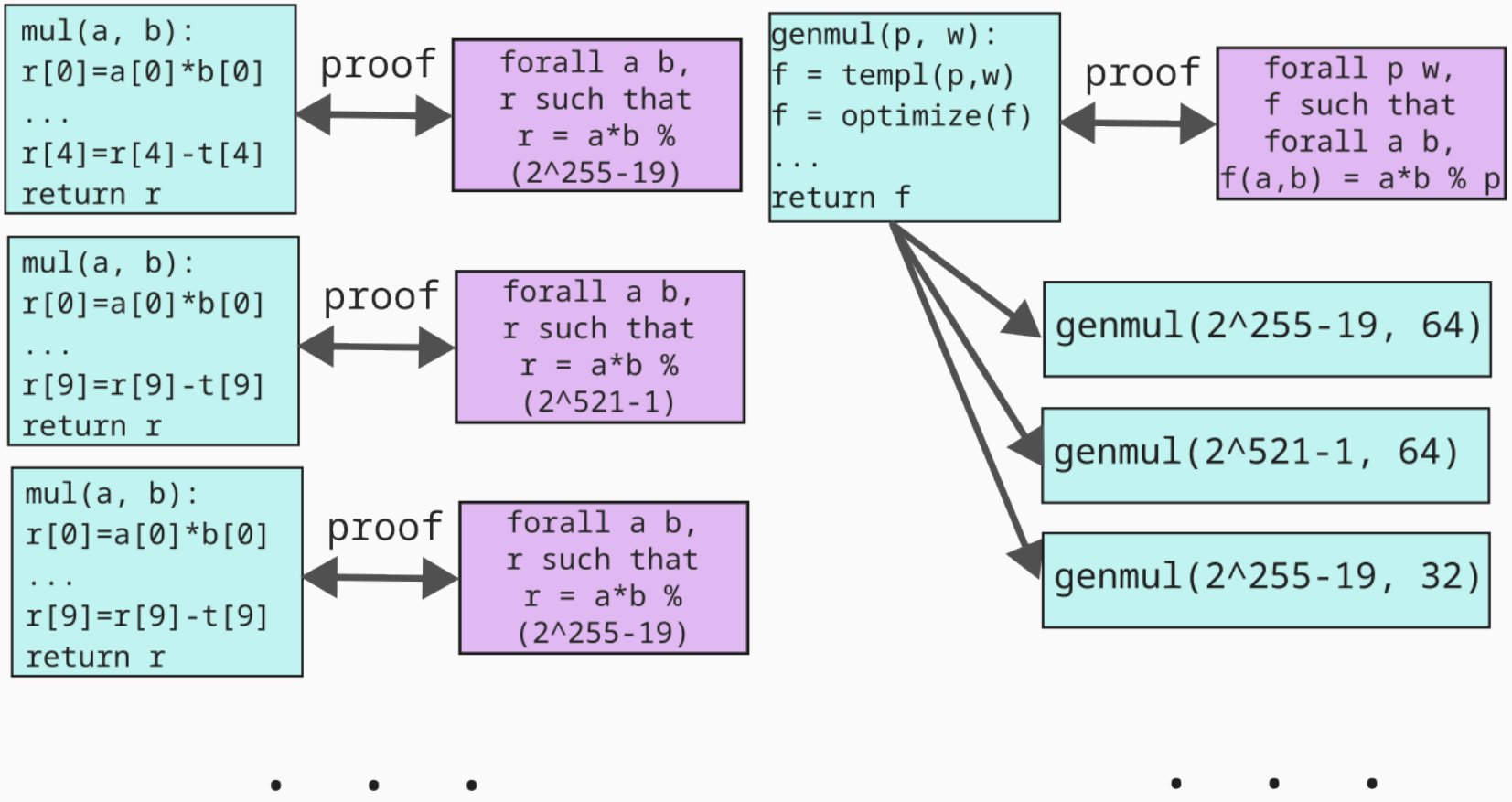

1 2 3 4 5 6 7 8 9 []= Reference.v - GNU Emacs at jinjur dwm-6.5
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* states that a [nat] is in the range [low...high) *)
Definition in_range (low high : nat) : nat -> Prop :=
fun x => x < high /\ low <= x.
(* states that we can expand the range and know that it
still holds everything in the original range *)
Lemma loosen_range :
forall low high new_low new_high,
high <= new_high ->
new_low <= low ->
forall x,
in_range low high x ->
in_range new_low new_high x.
Proof.
cbv [in_range].
intros. destruct H1.
-:**- Reference.v 45% L857 (Coq Script(1-) +2 Holes)
1 goal (ID 14)
- low, high, new_low, new_high : nat
- H : high <= new_high
- H0 : new_low <= low
- x : nat
- H1 : x < high
- H2 : low <= x
x < new_high /\ new_low <= x
U:%%- *goals* All L10 (Coq Goals +2)
U:%%- *response* All L1 (Coq Response +2)


1 2 3 4 5 6 7 8 9 []= Reference.v - GNU Emacs at jinjur dwm-6.5
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* states that a [nat] is in the range [low...high) *)
Definition in_range (low high : nat) : nat -> Prop :=
fun x => x < high /\ low <= x.
(* states that we can expand the range and know that it
still holds everything in the original range *)
Lemma loosen_range :
forall low high new_low new_high,
high <= new_high ->
new_low <= low ->
forall x,
in_range low high x ->
in_range new_low new_high x.
Proof.
cbv [in_range].
intros. destruct H1.
split; lia.
U:%%- *goals* All L1 (Coq Goals +2)
No more goals.
-:**- Reference.v 45% L858 (Coq Script(0-) +2 Holes)
U:%%- *response* All L1 (Coq Response +2)

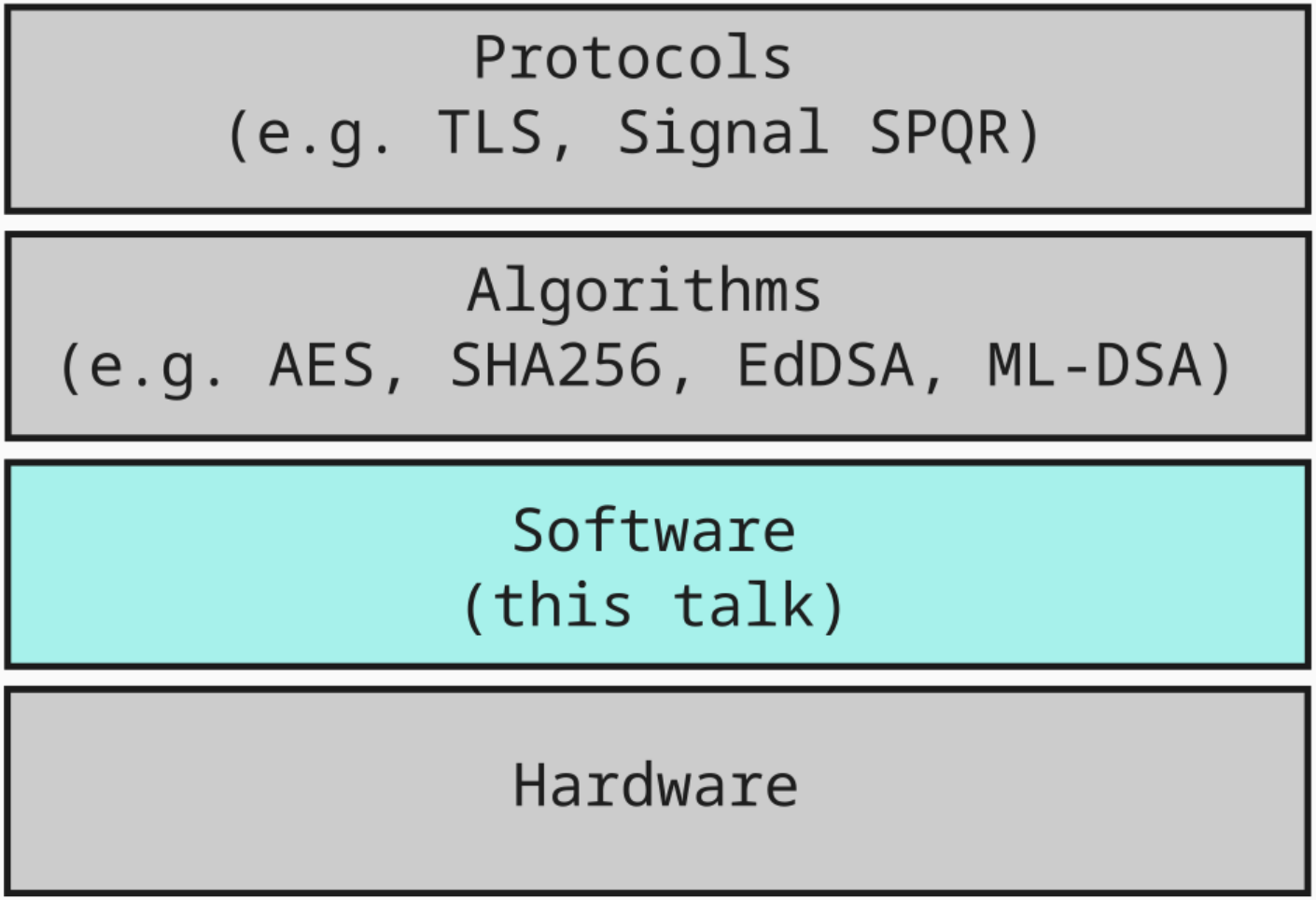

mul(a, b):
r[0]=a[0]*b[0]
...
r[4]=r[4]-t[4]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)


mul(a, b):
r[0]=a[0]*b[0]
...
r[4]=r[4]-t[4]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)
mul(a, b):
r[0]=a[0]*b[0]
...
r[9]=r[9]-t[9]
return r
proof
forall a b,
r such that
r = a*b %
(2^521-1)


mul(a, b):
r[0]=a[0]*b[0]
...
r[4]=r[4]-t[4]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)
mul(a, b):
r[0]=a[0]*b[0]
...
r[9]=r[9]-t[9]
return r
proof
forall a b,
r such that
r = a*b %
(2^521-1)
mul(a, b):
r[0]=a[0]*b[0]
...
r[9]=r[9]-t[9]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)


mul(a, b):
r[0]=a[0]*b[0]
...
r[4]=r[4]-t[4]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)
mul(a, b):
r[0]=a[0]*b[0]
...
r[9]=r[9]-t[9]
return r
proof
forall a b,
r such that
r = a*b %
(2^521-1)
mul(a, b):
r[0]=a[0]*b[0]
...
r[9]=r[9]-t[9]
return r
proof
forall a b,
r such that
r = a*b %
(2^255-19)
genmul(p, w):
f = templ(p,w)
f = optimize(f)
...
return f
proof
forall p w,
f such that
forall a b,
f(a,b) = a*b % p
genmul(2^255-19, 64)
genmul(2^521-1, 64)
genmul(2^255-19, 32)

Formal methods are also useful above and below the level of code.



Formally verified code is everywhere.

Formally verified code is everywhere.

Messengers, tools. . .

OpenSSL
Cryptography and SSL/TLS Toolkit

OpenSSH

Formally verified code is everywhere.

Messengers, tools. . .

OpenSSL
Cryptography and SSL/TLS Toolkit

. . . browsers. . .

Formally verified code is everywhere.

Messengers, tools...

OpenSSL
Cryptography and SSL/TLS Toolkit

...browsers...

...cloud platforms...

Formally verified code is everywhere.

Messengers, tools...

OpenSSL
Cryptography and SSL/TLS Toolkit

...browsers...

...cloud platforms...

...and popular or standard libraries.

Questions?