



VANDERBILT UNIVERSITY
College of Connected Computing

Day 4 — Program & High-Assurance Verification

FMAIV: Formal Methods & AI-Assisted Verification

Taylor Johnson — Associate Professor of Computer Science, Computer Engineering & Electrical Engineering; Associate Dean for Graduate Education, College of Connected Computing · taylor.johnson@vanderbilt.edu · taylortjohnson.com

Ben Wooding — Postdoctoral Scholar, Institute for Software Integrated Systems · ben.wooding@vanderbilt.edu · woodingben.com

Vanderbilt University

Day 4 of 4

Day 4 — Programs & the Frontier

CBMC, Cryptol, SAW, and where the field is going

Run it live (no install): Codespaces (full toolset) or Colab — today's notebook `04_day4_program_verif`, plus the frontier NN notebooks `05_nn_robustness` · `06_nn_mnist`. All materials: <https://github.com/ttj/fmaiv>

Where we are

Three views of the same counter so far:

Day	Model	Guarantee
1	Z3 / SMT formula	no counterexample of length $\leq N$
2	nuXmv / SMV	no counterexample, ever (finite state)
3	Lean / induction	machine-checked proof, any bound

Today: the same ideas, applied to **real code**.

Fresh from yesterday: an AI-discovered theorem, kernel-checked in a week

Day 3 closed on the **AI proposes, the kernel disposes** thesis — PFR, AlphaProof, DeepSeek-Prover. **News from this week:**

- **The result.** OpenAI's research stack announced a **disproof of Paul Erdős' planar unit-distance conjecture** (May 20, 2026) — a long-open extremal-geometry problem we flagged on Day 3 as *generated but not yet formalized*.
- **Then.** Logical Intelligence's **Aleph Prover** delivered a public **Lean 4** formalization (blog dated 2026-05-29) — i.e. **~9 days end-to-end** — with professional mathematicians (Kevin Buzzard, Sergey Galkin, others) validating along the way.
- **Scale.** **~33,000 lines** of Lean, **~252 Mathlib modules**, conditional on **two** explicit class-field-theory inputs (Golod–Shafarevich + Shafarevich's relation-rank bound) — reduced from ~30 open dependencies at the start.
- **Provenance.** Blog: <https://logicalintelligence.com/blog/aleph-prover-erdos-disproof-lean-4-formal-methods>; repo: <https://github.com/logicalintelligence/erdos-unit-distance> (Apache 2.0; Lean 4.29.1; CI runs `lake build` + external `leanchecker`).

The Day-3 thesis, *uncomfortably literal*: an AI generated a candidate proof; a Lean **kernel-checked** artifact certified it in ~9 days. For comparison, **PFR** (also flagged on Day 3) took ~3 weeks with a Blueprint-coordinated team — the cycle time is shrinking. That cadence is the Day-4 frontier we'll close on in L3.

Today's roadmap

Block	Topic
L1	Bounded model checking of C with CBMC
L2	Bit-precise specs with Cryptol ; C↔spec equivalence with SAW
L3	The frontier: neural-network verification + industrial deployments

Learning objectives for Day 4

By the end of today you will be able to:

- Frame a C program as a transition system on memory states.
- Write a **CBMC** (C Bounded Model Checker) harness with nondeterministic inputs and read its counterexample.
- Write a small **Cryptol** spec and discharge a property with `:prove`.
- Explain how **SAW** (Software Analysis Workbench) proves a C implementation equivalent to a Cryptol spec.
- Locate the field's frontier: NN verification, and where FM is deployed in industry.

L1 — Bounded model checking of C with CBMC

Programs are transition systems

A C program's **state** is:

(program counter, valuation of every variable, heap, stack)

and its **transition** is "execute one statement."

This is the same $T = (S, S_0, \rightarrow)$ from Day 1 — the state space is just much bigger.

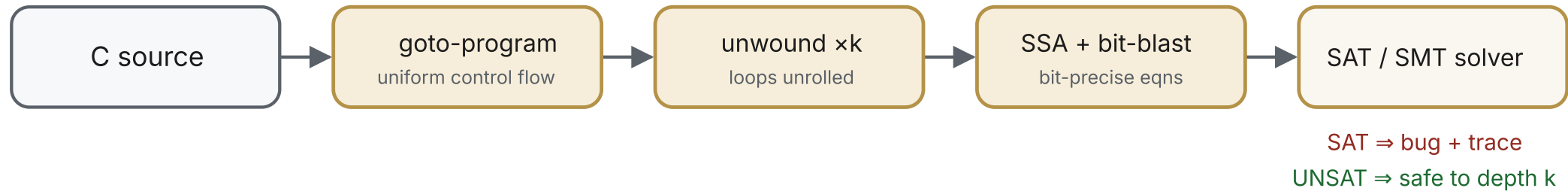
What CBMC actually does

C source → goto-program → unwound goto-program → SAT/SMT formula → solver

1. Parse and simplify C into a control-flow "goto-program."
2. **Unwind** loops a fixed number of times.
3. Encode every path as one big bit-precise formula.
4. Ask the solver whether any path violates an assertion.

Same idea as Day 1 — CBMC turns C into a satisfiability query (it bit-blasts to **SAT** by default; `--smt2` `--z3` emits SMT for Z3 instead).

The pipeline, stage by stage



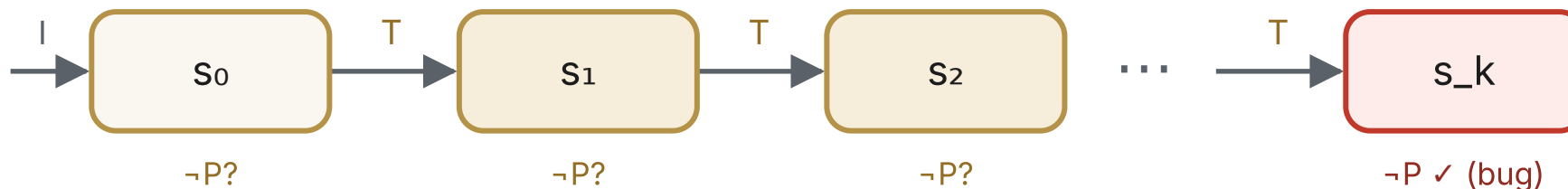
Caption: each CBMC stage lowers the program one level closer to a Boolean/bit-vector formula a solver can decide.

Stage	What it does
goto-program	rewrite all control flow (<code>if</code> , <code>for</code> , <code>while</code> , <code>?:</code>) into one uniform form: guarded <code>goto</code> s. Now there are no loops “shapes,” just jumps.
unwind	replace each loop by <code>k</code> copies of its body (Day-1’s bound, applied to code).
SSA	rename every variable so it is assigned once (<code>x</code> , then <code>x_1</code> , <code>x_2</code> , ...). A program becomes a list of equations.
bit-blast	model each <code>int</code> as 32 individual bits; <code>+</code> , <code>&</code> , <code><<</code> become Boolean circuits — <i>bit-precise</i> .
solve	hand the formula to a SAT solver (MiniSat/CaDiCaL) by default, or to SMT (Z3/cvc5 via <code>--smt2</code>). SAT = a bug + trace; UNSAT = safe to depth <code>k</code> .

The safety formula CBMC builds

For a loop unrolled to depth k , CBMC asks the solver one question:

$$\underbrace{I(s_0)}_{\text{start state}} \wedge \underbrace{\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})}_{k \text{ steps of the program}} \wedge \underbrace{\bigvee_{i=0}^k \neg P(s_i)}_{\text{some step breaks } P}$$



Caption: bounded model checking unrolls the transition relation k times and asks whether the property can fail at any step.

- I = the **initial** condition; T = "execute one statement"; P = the **property** (your `assert`, plus the built-in checks).
- **SAT** (a solution exists) $\Rightarrow$ there is a real run reaching a bad state — the counterexample.
- **UNSAT** (no solution) $\Rightarrow$ no run of length $\leq k$ violates P — **safe to depth k** .

This is *exactly* the Day-1 bounded-model-checking encoding — only T is now extracted from C source automatically.

Loop unwinding

```
cbmc counter.c counter_check.c --unwind 26 --unwinding-assertions
```

- `--unwind N` — unfold each loop `N` times.
- `--unwinding-assertions` — CBMC checks `N` was large enough and **fails loudly** if not. (CBMC 6 enables this by default; we write it explicitly to be clear.)
- Off-by-one: a loop running `BOUND = 25` times needs `--unwind 26` — one extra for the termination check.

The harness: nondeterminism + assertions

```
bool nondet_bool(void);    // CBMC: returns an unconstrained value

int main(void) {
    struct state s = { .mode = MODE_OFF, .x = 0 };
    for (int i = 0; i < BOUND; ++i) {
        bool press = nondet_bool();    // environment picks each step
        counter_step(&s, press);
        assert(s.x ≤ 10);              // the property
    }
}
```

- `nondet_*`() = the SMV “free input” idiom, in C.
- `__CPROVER_assume(c)` constrains inputs; `assert(c)` states the property.

Live demo: the counter in C

```
void counter_step(struct state *s, bool press) {  
    if (s->mode == MODE_OFF && !press) { /* stay off */ }  
    else if (s->mode == MODE_OFF && press) { s->mode = MODE_ON; }  
    else if (s->mode == MODE_ON && !press && s->x < COUNT_MAX) { s->x++; }  
    else if (s->mode == MODE_ON && (press || s->x ≥ COUNT_MAX)) {  
        s->mode = MODE_OFF; s->x = 0;  
    }  
}
```

Same four guards as SMV `next(...)`, Lean `next`, Z3 `step()`. (this is `counter.c` + `counter_check.c`)

```
cbmc counter.c counter_check.c --unwind 26 --unwinding-assertions  
→ VERIFICATION SUCCESSFUL
```

Reading a CBMC counterexample

Weaken the assertion to `s.x < 10` and re-run:

```
[main.assertion.1] file counter_check.c line 57 assertion s.x < 10: FAILURE  
  
State ... counter.c function counter_step  
  prev_x = 9  
State ... counter.c  
  s.x = 10  
Violated property: s.x < 10
```

CBMC prints the **exact** state sequence — and the precise `press` inputs — that drive `x` to 10.

Reading a trace step by step

A CBMC trace is a **list of state updates** in execution order. Read it like a debugger replay:

```
State 12 file counter_check.c line 49      s = { .mode = OFF, .x = 0 }      ← initial
State 18 file counter_check.c line 52      press = TRUE                ← input chosen
State 24 file counter.c line 41             s.x = 1                    ← step 1
      : (CBMC chooses press, x climbs 1,2,...,9)
State 71 file counter_check.c line 52      press = FALSE
State 77 file counter.c line 41             s.x = 10                    ← step 10
Violated property: counter_check.c line 57  s.x < 10
```

- Each **State ...** line is one assignment; the **inputs** (**press = ...**) are the part you replay.
- The last block names the **violated property** and its source line — the smoking gun.
- Replaying just the **press** values in a debugger reproduces the bug deterministically.

For a more readable view of the same trace — input column highlighted, **x** shown as a bar chart, transitions annotated — pipe **cbmc --trace** through **scripts/cbmc-trace-replay.py**:

```
sed 's/s\.x ≤ 10/s.x < 10/' counter_check.c > /tmp/strict.c
cbmc counter.c /tmp/strict.c --unwind 26 --unwinding-assertions --trace \
| python3 scripts/cbmc-trace-replay.py
```

Worked example: `array_max` + its counterexample

```
int array_max(const int *a, int n) {  
    int m = a[0];  
    for (int i = 1; i < n; ++i)  
        if (a[i] > m) m = a[i];  
    return m;  
}
```

Harness: fill `a[0..4]` with `nondet_int()`, then assert the **full spec** of “maximum”:
`m ≥ a[i]` for all `i` **and** `m` equals some `a[i]`.

```
cbmc array_max.c array_max_check.c --unwind 6 --unwinding-assertions  
→ VERIFICATION SUCCESSFUL
```

Now inject a bug — change the loop to `i < n - 1` (skips the last element):

```
[main.assertion.1] line 37 assertion m ≥ a[i]: FAILURE  
  a[0]=0 a[1]=0 a[2]=0 a[3]=0 a[4]=1      ← last element is the biggest...  
  m = 0                                     ← ...but the loop never looked at it  
Violated property: m ≥ a[i]
```

CBMC hands you the *smallest* array that exposes the off-by-one. (*this is*
`array_max.c` + `array_max_check.c`)

Worked example: the binary-search overflow

The famous JDK bug (publicized 2006): the **midpoint** of a binary search.

```
int mid = (lo + hi) / 2;    // BUG: lo + hi can overflow int
int mid = lo + (hi - lo) / 2; // SAFE: never overflows
```

- For large indices, `lo + hi` exceeds `INT_MAX` and **wraps to negative** — then `a[mid]` is an out-of-bounds read.
- CBMC's default overflow + bounds checks catch this *without any assertion you write*:

```
[binsearch.overflow.1] arithmetic overflow on signed + in lo + hi: FAILURE
```

The overflow-safe form verifies clean. Same class of bug as **Ariane 5** (Day 1) — a conversion/arithmetic overflow that testing missed for years. (*this is `binsearch.c` + `binsearch_check.c`*)

What CBMC catches by default

Even with no assertions you write, CBMC checks for:

- integer **overflow** (the Ariane 5 bug),
- array **out-of-bounds**,
- **pointer safety** (null / invalid dereference, dereference of freed memory) via `--
pointer-check`,
- **division by zero**.

Each is a built-in assertion; the counter run shows dozens of `SUCCESS` lines for these.

CBMC as a unit-test generator: `--cover`

So far we used CBMC to *prove* `assert` s. The **same solver** can do the opposite — *find* an input for every coverage goal you name:

```
cbmc gcd.c gcd_check.c --cover branch --unwind 11 # one input per branch
cbmc gcd.c gcd_check.c --cover decision --unwind 11 # one input per decision outcome
cbmc gcd.c gcd_check.c --cover mcdc --unwind 11 # MC/DC (DO-178C avionics-grade)
cbmc gcd.c gcd_check.c --cover location --unwind 11 # one input per source line
```

For **every** unreachable goal CBMC prints **FAILED** (dead code); for every reachable goal it prints **SATISFIED** along with the **concrete inputs** that hit it — exactly what a unit-test runner needs (CPROVER manual, `test-suite generation` and `standard checks`; tool source: <https://github.com/diffblue/cbmc>).

Coverage criterion	Captures
<code>location</code>	every source line is reached
<code>branch</code>	every <code>if / while / for</code> decision goes both ways
<code>decision</code>	every <i>compound</i> boolean expression evaluates true and false
<code>mcdc</code>	every atomic condition is independently shown to affect the outcome — the standard for DO-178C Level A avionics

The shift: “prove the assertion” and “generate one test per goal” are the same SAT/SMT query with different objective formulas. One tool, two modes.

Worked example: branch coverage for GCD

The Euclidean GCD function from Day 2 (`gcd_01.smv`), now in C as `gcd.c` + `gcd_check.c` :

```
int gcd(int a, int b) {  
    while (b  $\neq$  0) { int t = b; b = a % b; a = t; }  
    return a;  
}
```

One harness, two CBMC runs. *Property* mode proves the spec for every `(a, b)` in $[0, 20]^2$; *coverage* mode generates one unit-test input per branch:

```
$ cbmc gcd.c gcd_check.c --unwind 11 --unwinding-assertions  
→ VERIFICATION SUCCESSFUL                                     # property mode  
  
$ cbmc gcd.c gcd_check.c --cover branch --unwind 11  
[gcd.coverage.1]      line 28 function gcd entry point: SATISFIED    # 10 goals,  
[gcd.coverage.2]      line 28 block 1 branch false:      SATISFIED    # one  
[gcd.coverage.3]      line 28 block 1 branch true:       SATISFIED    # SATISFIED line  
[main.coverage.1-7]   line 26-43 ...                      SATISFIED    # per branch  
** 10 of 10 covered (100.0%)                                # ← every branch hit
```

Each **SATISFIED** line carries the **concrete (a, b) pair** the solver found — that pair becomes one unit test. **Counter example** (`counter_check.c`) works the same way: `--cover branch` enumerates the press sequences that hit each of `counter_step`'s four guards.

The complement to CBMC: deductive verification

CBMC is **bounded** — it checks every run up to depth **N**. The other major style proves correctness for **all** inputs and **all** depths — the price is that *you* supply the invariants.

- **Hoare triple** $\{P\} c \{Q\}$: if precondition **P** holds and **c** terminates, postcondition **Q** holds.
- The creative work is the **loop invariant** (true on entry, preserved each iteration); add a **variant** (a measure that strictly decreases) to also prove **termination** — *partial* vs *total* correctness.
- You write **requires** / **ensures** / **invariant**; an **SMT solver discharges** the proof obligations — Day 1's Z3, finally cashed out on real code.
- Tools: **Dafny** (gentlest, web IDE), **Verus** (verifies **Rust** — squarely on the AI-generated-code thesis), **Frama-C**, **Why3**, **Viper**.

Two complementary guarantees: CBMC finds bugs *fast* (bounded, push-button); deductive verification proves *everything* (unbounded — you write the invariant).

L1 recap

- A C program is a **transition system on memory**; CBMC builds it from source.
- CBMC = C → goto-program → unwound → SAT/SMT → solver (Day 1's decision procedure, automated; SAT by default, Z3 optional).
- Harness with `nondet_*` + `assert` + `__CPROVER_assume`; `--unwind N` bounds loops (CBMC 6 checks `N` is big enough by default).
- Default checks catch overflow, bounds, null deref, division-by-zero for free.



Break

We resume after the break with Cryptol and SAW.

L2 — Cryptol + SAW

Why a separate specification language

CBMC checks code against inline assertions. But often you want:

- a **reference spec** independent of any implementation, and
- a proof that a fast/tricky implementation **equals** the spec on all inputs.

"Spec once, verify many implementations." This is **Cryptol** (Galois, first published 2003) + **SAW**.

What is Cryptol?

A small functional language for writing **executable specifications** of algorithms that work on bits and bytes — built by Galois, first published 2003, originally for cryptography.

- You write *what* the algorithm computes, at the bit level — not *how* to make it fast.
- The same file is both **runnable** (test it) and **provable** (`:prove` checks it for *all* inputs).
- Think “executable math for bit-vectors”: if you can write AES on a whiteboard, you can write it in Cryptol almost line-for-line.

Why it matters today: it's the cleanest place to see “spec = proof target,” and it's how AWS verifies real crypto (later this block).

Cryptol types are sizes — read `[n]T` as “n of T”

Every type carries a size, checked at compile time. That bit-exactness is why Cryptol is the language for crypto and hardware specs.

You write	You say	It means
<code>Bit</code>	“bit”	one bit (a Boolean)
<code>[8]</code>	“8 bits”	an 8-bit word (= <code>[8]Bit</code>)
<code>[16][8]</code>	“16 of 8-bit”	16 bytes (a 16-byte buffer)
<code>[4][8]</code>	“4 of 8-bit”	4 bytes
<code>(A, B)</code>	“a pair”	a tuple of an <code>A</code> and a <code>B</code>
<code>A → B → C</code>	“a function”	takes an <code>A</code> and a <code>B</code> , returns a <code>C</code>

- The **number is the length**, and it’s **part of the type** — so the compiler always knows every size.
- That’s why `[8] + [8]` type-checks but `[8] + [4]` is a compile error.
- Multiple arrows = multiple arguments: `[8] → [N][8] → [N][8]` takes a key byte **and** a message.

Cryptol by example: a shift cipher

```
type N = 16                                     // message length, in bytes ([N][8] = N bytes)
encrypt : [8] → [N][8] → [N][8]               // key byte → 16 bytes → 16 bytes
encrypt k msg = [ c + k | c ← msg ]           // add the key to every byte (+ is mod 256)

decrypt : [8] → [N][8] → [N][8]
decrypt k ct  = [ c - k | c ← ct ]

property roundtrip k msg = decrypt k (encrypt k msg) == msg
```

- `type N = 16` names a compile-time size — usable anywhere a length is needed.
- `[ c + k | c ← msg ]` is a **comprehension**: “for each byte `c` drawn from `msg` (`←`), produce `c + k`” — like building a new array by transforming each element.
- `+` / `-` on `[8]` are arithmetic **mod 256** — the width lives in the type.
- a **property** is a claim to check for **all** inputs. (*this is `caesar.cry`*)

:check , :prove , :sat

```
caesar> :check roundtrip          -- fast: random testing, no solver
Using random testing.
passed 100 tests.
caesar> :prove roundtrip          -- exhaustive: ALL keys × ALL 16-byte messages
Q.E.D.
caesar> :sat \k msg → decrypt k (encrypt k msg) ≠ msg
Unsatisfiable
```

- **:check p** — quick random sanity test (no solver), great before the real proof.
- **:prove p** — “**p** holds for **all** inputs”; **Q.E.D.** = proved exhaustively.
- **:sat e** — find inputs making **e** true; **Unsatisfiable** = none exist. (**\k msg → e** is an anonymous function of **k** and **msg**.)

:prove p is exactly “**¬p** is unsatisfiable” — Day 1’s validity ↔ unsat-of-negation, now over fixed-width bit vectors. Cryptol talks to **Z3** by default (through its SBV backend). (Run the **caesar_starter.cry** stub and **:prove roundtrip** returns a **Counterexample** instead.)

The counter in Cryptol (fifth encoding)

```
type State = (Bit, [4])           // (mode, x): [4] = 0..15, headroom to ask "x = 11?"

step : State → Bit → State
step s press = ...                // four guards, mirrors counter.c

property bounded_invariant (presses : [25]Bit) =
  foldl1 (&&) True [ inv s | s ← states ]      // states = the 25-step trajectory

property inductive_invariant (s : State) (press : Bit) =
  if inv s then inv (step s press) else True    // mirrors Lean step
```

```
:prove bounded_invariant    → Q.E.D.    (~0.2s)
:prove inductive_invariant   → Q.E.D.    (~0.02s)
```

(`foldl1 (&&) True xs` folds "and" across a list — "are all of `xs` true?"; `states` is the trajectory the file builds from `presses` .) (*this is `counter.cry`*)

SAW: tying Cryptol to a C implementation

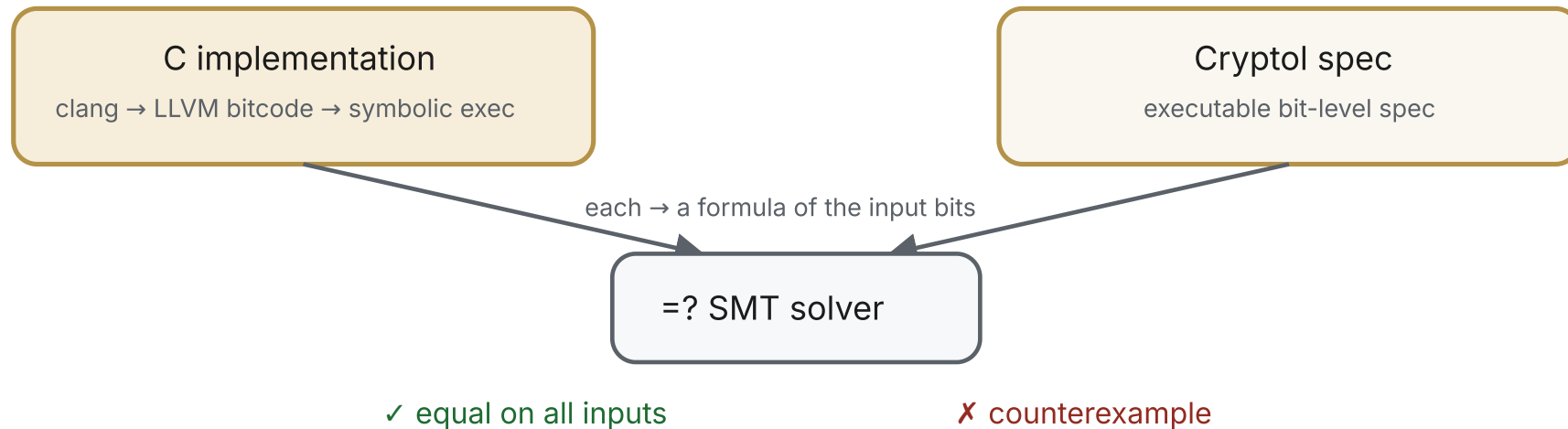
The SAW example is **popcount** (count the set bits in a word): a clean Cryptol spec plus an optimized C implementation. SAW proves they compute the same function.

```
clang -c -emit-llvm -O0 -o popcount.bc popcount.c    # C → LLVM bytecode
saw popcount.saw                                     # prove C = Cryptol spec
```

```
Proof succeeded! popcount_loop
```

SAW extracts a symbolic model of each C function from LLVM bytecode and proves it **equivalent** to the Cryptol spec, over all inputs, via SMT.

How SAW works: symbolic execution



Caption: SAW turns both the compiled C and the Cryptol spec into formulas over the same symbolic input, then asks the solver if they are equal for every input.

1. **Compile C to LLVM bitcode** (`clang -emit-llvm`) — the same intermediate form the compiler optimizes; SAW reads *that*, not the text.
2. **Run the function on a symbolic input** — a variable standing for *all* bytes at once, not one number. SAW threads it through the code and gets the output **as a formula** of the input bits.
3. **Compare to the spec.** The Cryptol spec is already a formula. SAW asks the solver: “*are these two formulas equal for every input?*”
4. **UNSAT of the difference ⇒ equivalent.** Same validity = unsat-of-negation logic as Days 1-3.

A `.saw` script, annotated

```
import "popcount.cry";           // bring in the Cryptol spec
m ← llvm_load_module "popcount.bc"; // load the compiled C

llvm_verify m "popcount_loop" [] false (do {
  x ← llvm_fresh_var "x" (llvm_int 8); // a SYMBOLIC 8-bit input (all bytes at once)
  llvm_execute_func [llvm_term x];     // call the C function on it
  llvm_return (llvm_term {{ (zero : [4]) # popcount_simple x }}); // result = spec (pad [4]→[8])
}) z3;                                // discharge with the z3 solver
```

- `llvm_fresh_var` = “an input standing for **every** value” (CBMC’s `nondet_*`, in SAW).
- `{{ ... }}` switches into **Cryptol** — `popcount_simple x` is the reference answer.
- The script reads as a contract: *given* this symbolic input, *after* calling the function, the **return equals the spec**. (*this is* `popcount.saw`)

The popcount example shape

Spec and implementation of the *same* function — “count the 1-bits”:

```
popcount_simple : [8] → [4]           // Cryptol SPEC: sum the bits
popcount_simple x = sum [0 # [b] | b ← x]
```

```
uint8_t popcount_loop(uint8_t x) {      // C IMPLEMENTATION
    uint8_t count = 0;
    for (int i = 0; i < 8; ++i)
        count += (x >> i) & 1U;         // tally bit i
    return count;
}
```

- One width mismatch to bridge: C returns `[8]`, the spec returns `[4]`. The script pads with four zero bits — `(zero : [4]) # popcount_simple x` — so the types line up.
- SAW proves these are the **same function on all 256 inputs**: *Proof succeeded!*
popcount_loop. (this is `popcount.cry` + `popcount.c`)

Where SAW shines — and where it struggles

- **Shines:** crypto (AES, SHA, P-256), parsers, format validators, fixed-shape loops.
- **Struggles:** data-dependent loops (e.g. Kernighan's `while (x  $\neq$  0)`) need explicit bounds — symbolic execution can't always tell when to stop.

We grade only the counted-loop variant in SAW; the two Cryptol definitions are linked by `:prove`. A clean architecture beats fighting the tool.

L2 recap

- **Cryptol** = bit-precise spec language; `:prove` / `:sat` discharge properties over all inputs via SMT.
- **SAW** proves a C/LLVM implementation **equivalent** to a Cryptol spec.
- The counter's bounded *and* inductive invariants both fall to Cryptol `:prove`.
- Match the obligation to the tool — fixed loops for SAW, `:prove` to chain implementations.



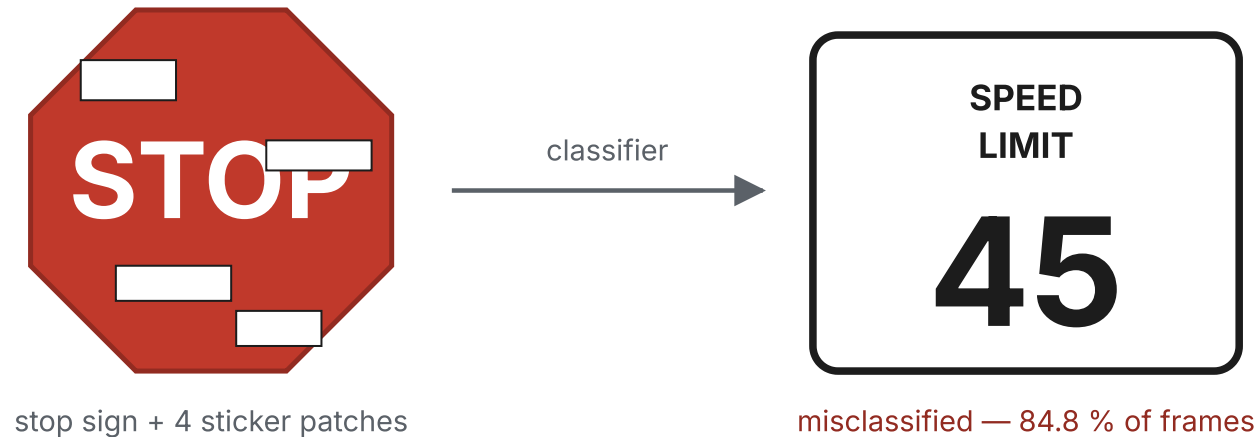
Break

We resume after the break with the frontier — neural-network verification and industrial FM.

L3 — The frontier

A stop sign + stickers = "Speed Limit 45"

A handful of **printable stickers** — not paint, not graffiti — on a real road sign make a deployed deep classifier read **Speed Limit 45** in **84.8 %** of drive-by video frames (Eykholt et al., *Robust Physical-World Attacks on Deep Learning Visual Classification*, CVPR 2018; <https://arxiv.org/abs/1707.08945>).



- **Physical**, not pixel-only: stickers printed, applied to the *actual* sign, photographed on a moving vehicle at varying distances and angles.
- The perturbation **survives a real camera + lighting pipeline** — no hand-crafted noise, no white-box gradient hidden in the JPEG.
- Why this drives *formal* robustness: testing some images cannot rule out attackers like this; we want a **guarantee over a whole neighborhood of inputs around the true stop sign**.

Verifying neural networks

The problem: given a trained network f and an input region R ,

$\forall x \in R, f(x)$ still classifies correctly (robustness).

Here R is usually an ℓ_∞ **ball** — every input within ε of a sample (each coordinate nudged by $\leq \varepsilon$). *Robustness* = small input changes never flip the output class. Hard because f is **non-convex** (the safe region isn't a simple shape, so checking corners is not enough), **non-linear**, and has millions–billions of activations.

A network is just a function

For verification, strip away the training story. A trained feed-forward / convolutional net is a **fixed mathematical function** $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$:

$$f(x) = W_L \sigma(\cdots \sigma(W_1 x + b_1) \cdots) + b_L$$

- W_i = a **weight matrix**, b_i = a **bias vector** — just numbers fixed at training time.
- Each layer = **matrix multiply, add bias, apply** σ (the *activation*). Repeat for L layers.
- σ is usually **ReLU** (rectified linear unit): $\sigma(z) = \max(0, z)$ — “pass positives through, zero out negatives.”
- For digit recognition: $n = 256$ pixels in, $m = 10$ class scores out; the answer is the **argmax** (the highest-scoring class).

So verifying a network = reasoning about a (big, non-linear) function — the same object we have reasoned about all week.

The robustness property, precisely

Given a correctly-classified input x_0 (say, an image of a **2**), demand:

$$\forall x. \|x - x_0\|_\infty \leq \varepsilon \Rightarrow \arg \max f(x) = \arg \max f(x_0)$$

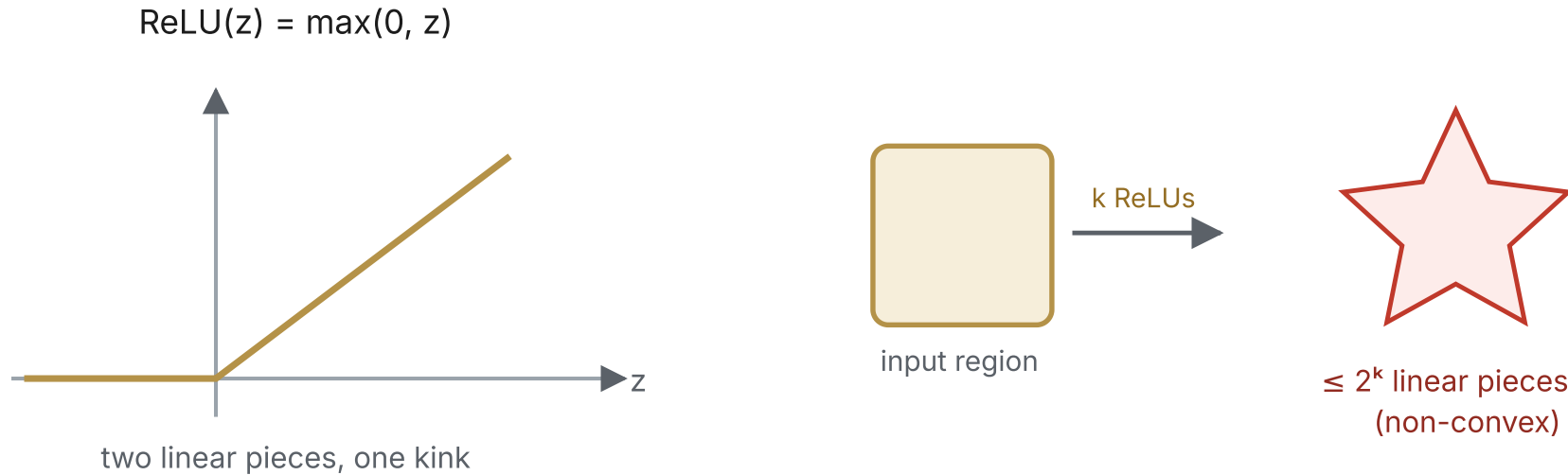
- $\|x - x_0\|_\infty \leq \varepsilon$ = the **ℓ_∞ ball**: every pixel may move by at most ε (brightness nudged up/down a little).
- The claim: **every** such nudged image still classifies as a **2** — no small perturbation flips the label.
- An x that *does* flip it is an **adversarial example** — the counterexample of NN verification.

This is a $\forall$ -over-a-region property — Day 1's "no bad input exists," now over a continuous box of images.

ℓ_∞ robustness is the canonical property, not the only one: the same machinery handles other input regions (rotations, NLP word-substitutions), output **safety/reachability** for control networks (ACAS Xu), and monotonicity/fairness — any "input-set $\Rightarrow$ output-set" claim.

Why this is hard: ReLU explodes the cases

Each ReLU neuron is **piecewise-linear** — two linear pieces with a kink at 0:



Caption: each ReLU splits the input region into an “active” and “inactive” half; with k neurons that is up to 2^k linear regions.

- Per neuron, the input region splits into **active** ($z > 0$) and **inactive** ($z \leq 0$) parts.
- With k ReLU neurons, that is up to 2^k **linear regions** — exponential blow-up.
- The safe set is therefore **non-convex** (not a simple box/ball) and **non-linear** overall.
- Exact ReLU verification is worst-case intractable: **finding an adversarial example is NP-complete** (Katz et al., *Reluplex*, CAV 2017); **proving robustness — its complement — is co-NP-complete**.

Two solver families

	(a) Bound propagation + branch-and-bound	(b) Set-based reachability
Idea	compute linear lower/upper bounds on outputs; split cases when too loose	propagate a set through every layer; check the output set
ReLU handling	relax each ReLU to a linear envelope, tighten by branching	split the set at each kink; track exact/over-approx regions
Representative	α, β-CROWN (VNN-COMP winner)	NNV (our group) with star sets (Tran et al., FM 2019)
Verdict	bounds exclude bad outputs $\Rightarrow$ robust	output set avoids the bad region $\Rightarrow$ robust

Both certify robustness the same way: **UNSAT** $\Rightarrow$ no in-region input reaches a bad output.

A second axis — completeness vs. cost: *incomplete* methods (bound propagation) are fast but may answer “**unknown**”; *complete* methods add **branch-and-bound** to always decide, at higher cost. Winners run cheap-first, branch only where needed.

Reachability: push a set through the net

NNV's recipe (our group) — verification *as* reachability (Tran et al., *Star-Based Reachability Analysis of Deep Neural Networks*, FM 2019):

$$\text{inputs } R \xrightarrow{\text{layer 1}} \cdot \xrightarrow{\text{layer 2}} \dots \xrightarrow{\text{layer } L} \text{output set } f(R)$$

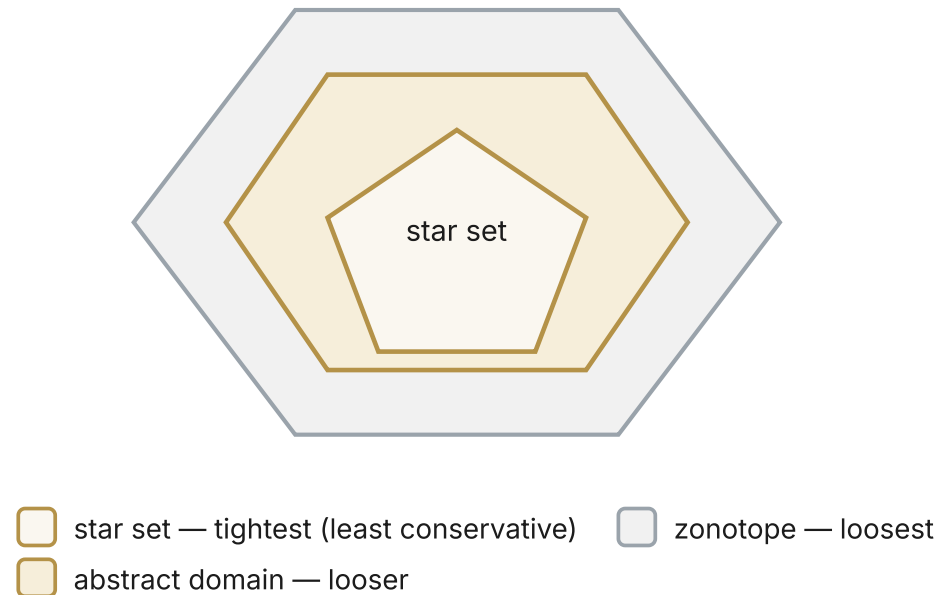
- **Affine layers are easy:** an affine map of a polytope (a flat-sided region like a box/polygon) is again a polytope (scale/rotate/translate a shape, get a shape). This handles every matrix-multiply + bias.
- **ReLU layers split:** the set may break into a **union of polytopes** (the 2^k blow-up) — the hard part. *Exact* reachability splits the set at each ReLU; the **approximate** star method over-approximates each ReLU (**sound, may be incomplete**) to scale to large nets.
- **Check at the end:** does $f(R)$ intersect the **unsafe** region (a different class scores higher)? **Empty intersection $\Rightarrow$ robust.**

The transition relation is *the network itself* — Day 1's reachability, with layers as steps.

Star sets: making it scale

The blow-up is real — so the *representation* of the set is everything. NNV uses **star sets**.

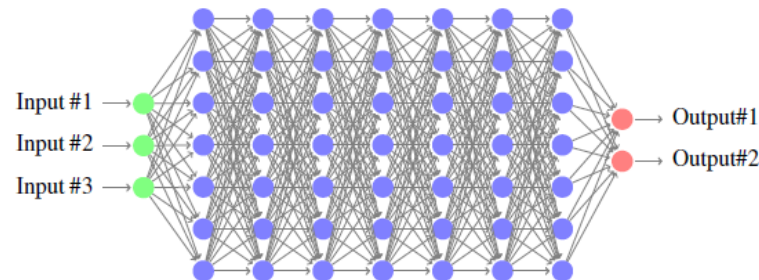
- A **star set** compactly encodes a polytope as a **center + basis vectors + a predicate** (linear constraints) — “this shape = these directions, subject to these inequalities.”
- Closed under exactly the two operations NN verification needs: **affine maps** (for layers) and **intersection with a half-space** (everything on one side of a plane) — for the safety check.
- vs. **zonotopes / interval / abstract-domain** representations: stars carry **far less over-approximation** — we report **10×–10,000× speedups** with **less conservatism** than DeepZ / DeepPoly / ReluVal (Tran et al., FM 2019).
- Less over-approximation = fewer **spurious** “maybe unsafe” verdicts (the false-positive problem from static analysis, again). The approximate star method stays **sound** (it never misses a real violation) but **may be incomplete** — those spurious verdicts are the price of scaling; the exact method splits each ReLU to recover completeness, at higher cost.



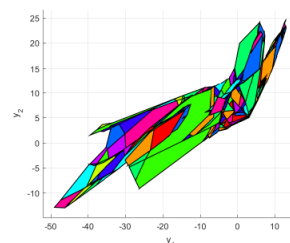
Caption: for the same true output set, looser representations (zonotopes) over-approximate badly; star sets stay tight.

Exact reachability in pictures: a 3-input / 2-output net

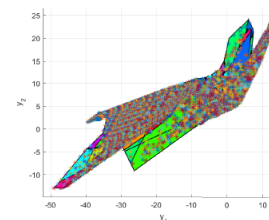
A canonical NNV demo — a small **3-input / 2-output ReLU MLP** (~5 hidden layers), reachable output set computed *exactly* with star sets (Tran et al., FM 2019).



A 3-input → 5 hidden layers → 2-output ReLU MLP.



Exact: every ReLU split produces a new polytope; the union is the true reachable set.



Same polytopes + ~10k random forward simulations. Every sample lands inside; no holes, no over-approximation.

- **Star sets keep the picture honest.** Each colored region is one piece of the *true* output set produced by an exact ReLU split. The simulations confirm soundness (every sample is contained) **and** tightness (no empty slack).
- Our auto_LirPA recreation in `notebook 05` / `compare_reachability.py` shows the **bound-propagation** complement: a *rectangle* (IBP / CROWN / α -CROWN) that contains this whole polygon — sound, cheaper, looser.

(Source: T. Johnson's NNV lab — the exact 3-input / 2-output `compareReachability` benchmark;
<https://github.com/verivital/nnv/tree/master/code/nnv/examples/Tutorial/NN/compareReachability>.)

MNIST robustness: set representations side by side

Scaling the picture to a real classifier — **MNIST handwritten digits** (10 classes, 28×28 inputs):



A sample of MNIST training images, 10 rows × 15 columns.

For each test image x_0 , **CROWN** / **ImageStar** / **α -CROWN** all answer the same question — *is every input in the ℓ_∞ ε -ball classified the same way?* — by propagating a **set** through the convolutional layers and projecting to **ten output intervals** $[lo_j, hi_j]$ (one per class). Robustness holds iff:

$$lo_{\text{true}} > \max_{j \neq \text{true}} hi_j$$

- **CROWN** (`auto_LiRPA`) — linear lower/upper bounds; ms-per-image on CPU. *Our colab demo.*
- **ImageStar** (NNV; Tran et al., **CAV 2020**) — exact CNN reachability through Conv + AvgPool + BatchNorm + ReLU (uses **exact** ReLU splitting + the star-set representation).
- **α -CROWN** / **β -CROWN** — bound propagation + **branch-and-bound** → *complete* verifier; the VNN-COMP MNIST-FC champion.

Live in our stack: `notebook 06` + `verify_fc.py` plot the **per-class CROWN intervals** for a single digit (the `verify_fc.m` analog from NNV) and compare CROWN vs IBP vs α -CROWN **certified accuracy** across 100 images.

(References: Tran et al., “Verification of Deep Convolutional Neural Networks Using ImageStars”, CAV 2020; MNIST dataset: <http://yann.lecun.com/exdb/mnist/>.)

NN verification in the wild

Real networks we have verified — beyond toy MLPs:

System	Network	Property verified
ACAS Xu	45 nets, 6×50 neurons each	collision-avoidance advisories stay correct in safe regions
VGG16/19 (ImageStar; Tran et al., CAV 2020)	16–19 layers, ~140M params, ImageNet 1000 classes	robust (sound over-approximation) to a bounded ℓ_∞ perturbation of an ImageNet image (≈ 10 min, 1 core)
CARLA / perception	conv nets on driving images	classification stable under ℓ_∞ image noise
Closed-loop CPS (Lopez et al., <i>NNV 2.0</i> , CAV 2023; Verisig: Ivanov et al., HSCC 2019)	net + plant dynamics (ACC, cruise control)	the <i>controlled system</i> stays safe over time

The frontier reach: from a 300-neuron advisory net to a 140-million-parameter image classifier, and from a bare network to a **network-in-the-loop** control system.

Where NN verification is being applied

A widening application surface — far beyond MNIST/CIFAR toys (sources: VNN-COMP 2020–2025 benchmark archive; NNV case-study log; Liu et al., *Algorithms for Verifying Deep Neural Networks*, Found. & Trends in Optimization, 2021):

Domain	What is being verified	Representative benchmarks / references
Image classification	local ℓ_∞ robustness of MLPs / CNNs / ResNets / ViTs	MNIST-FC, CIFAR-10/100, ImageNet (VGG-16/19 via ImageStar; Tran et al., CAV 2020), tinyimagenet , vit
Semantic / medical segmentation	per-pixel class stability of U-Nets	Carvana UNet (VNN-COMP 2022+); brain-MRI / lung-CT segmentation (Tran et al., FM 2021)
Speaker / audio recognition	invariance under bounded acoustic perturbations	Speaker-ID CNNs; VeriX explainability + verification on audio classifiers
Video classification	per-frame and short-horizon temporal stability of 3D-CNNs	UCF-style action-recognition robustness studies (Pal, Musau et al.)
Fairness in tabular ML	swaps in protected attributes never flip the label	FairSquare (Albarghouthi, OOPSLA 2017); DICE (Galhotra et al.); FairBoost
Aircraft collision avoidance	safe-advisory regions of policy nets	ACAS Xu (every VNN-COMP, 2020 →)
Autonomous-driving perception	sign-recognition + lane-detection invariance to noise	Traffic-Signs-Recognition (Erascu, Postovan; VNN-COMP'23); cctsd_b_yolo , metaroom
Closed-loop CPS / RL	system-level safety with the <i>network in the loop</i>	NNV 2.0 (Lopez et al., CAV 2023); SafeRL (AFRL); CartPole, LunarLander
Database / ML-for-systems	bounded-error cardinality estimators, learned indexes	nn4sys (Lin et al., VNN-COMP 2022+)
LLM / VLM guardrails	property checks on small open LMs and content filters	AWS Bedrock <i>Automated Reasoning</i> ; Shi et al., <i>Robustness Verification for Transformers</i> , ICLR 2020

Whenever the property fits the “**input region** $\Rightarrow$ **output region**” shape from Day 1, the bound-propagation / reachability machinery transfers.

What works, what's hard, what's open

Capability is not just “parameter count” — it is **layer × architecture × spec**. Distilled from the VNN-COMP 2020–2025 benchmark archive + our AAAI’26 lab (<https://vnn-comp.github.io/#aaai2026>):

	Routinely verifiable today	Hard / partial	Open frontier
Layers	Linear/Affine, Conv2D, ReLU, AvgPool, BatchNorm, ResNet skip-connections	MaxPool (per-neuron split), Sigmoid/Tanh (curved relaxations are loose), ConvTranspose, sigmoidal <code>nn4sys</code> heads	Self-attention / multi-head, LayerNorm, Softmax, LSTM/GRU recurrence, dynamic shapes
Architectures	feed-forward MLPs, CNNs, small/medium ResNets, small ViTs, branched DAGs	Recurrent control (LSTM in the loop), U-Nets at full Carvana resolution	Large transformers, mixture-of-experts, LLMs, diffusion models
Sizes (one-shot robustness)	≤ 10 M params with bound-prop + B&B; up to ~140 M (ImageStar / VGG-16/19) for reachability	100 M – 1 B image models with heavy conservatism	LLMs (1 – 70 B+), VLMs, full agent stacks
Specs / input regions	ℓ_∞ balls; safe-region reachability for control; halfspace output specs	ℓ_2 / ℓ_1 balls, rotations, brightness; discrete word-substitutions for NLP	semantic robustness across pixel-space and natural-language; “no jailbreak”
Verdict you can expect	sound + complete via branch-and-bound (α, β -CROWN) or exact star reachability	sound + incomplete — verifier returns unknown on the hard pieces	no mature <i>sound</i> verifier exists for the full frontier yet

Rule of thumb (2025): “feed-forward / piecewise-linear / ℓ_∞ / image scale” → solved; “recurrent / curved / discrete / language” → research.

VNN-COMP: the state of the art

Like SAT-/SMT-COMP, **VNN-COMP** is the field's annual scoreboard — 6 editions through 2025 (the latest run under the SAIV symposium, co-located with CAV), tools standardized on **ONNX** (the network) + **VNN-LIB** (the spec).

- **In routine reach:** ReLU MLPs, CNNs, ResNets, small transformers; ℓ^∞ robustness, reachability.
- **Winner 2021–2025 — five straight years: α, β -CROWN**, i.e. GPU-accelerated linear bound propagation + branch-and-bound — the configuration the strongest tools have converged on.
- **Still open:** large transformers / LLMs, recurrent nets, high-dimensional inputs, non- ℓ^∞ specs.

(Results & benchmarks: <https://vnn-comp.github.io/>; the annual competition reports are linked there.)

The frontier: from images to language & autonomy

Our grand challenge — “*Let’s verify ChatGPT*”: what would we even verify, and how? (Johnson, *Is Neural Network Verification Useful and What Is Next?*, Allerton 2025.)

- **Today’s reach:** verifiers handle **hundreds of millions of parameters** (e.g. ResNets; our ImageStar VGG-16/19 work reaches ~140M) for **ℓ_∞ robustness of image classifiers** — but parameter count alone isn’t a clean capability boundary (architecture, the spec, and completeness all matter as much as size).
- **The needed shift:** to **NLP** (sentiment, hate-speech, and **guardrail** models) and **vision-language-action (VLA)** robot policies — new architectures, new specs.
- **A realistic near-term target:** fully **open small language models** — Ai2’s **OLMo2-1B**, HuggingFace’s **SmolLM2-135M** — already near the scalability envelope, and the industry push to shrink models for cheap inference only helps.
- **Beyond one network: neuro-symbolic** systems — neuro-symbolic behavior trees (**BehaVerify**; Serbinowska & Johnson, SEFM 2022; *Formalizing Stateful Behavior Trees*, FMAS 2024) and neuro-symbolic automata (Sasaki, Lopez & Johnson, NeuS 2025) — compose NN verification with classical model checking; **NNV 2.0** now also covers CNNs, neural ODEs, RNNs, and binary nets.
- **Both directions:** foundation models *for* verification (draft specs, models, act as oracles) **and** verification *of* foundation models.

Transformer Explainer: see what we'd have to verify

What is *inside* the model when we say “verify ChatGPT”? The **Transformer Explainer** (Cho et al., 2024; arXiv:2408.04619) is a browser-based, live interactive of GPT-2 — watch the embedding, attention heads, MLP blocks, and the softmax token sampler update as you type.

- **Embedding + positional** — a bag of float vectors, layer-norm'd. *Robustness here is the easy part: bounded perturbations of a continuous vector.*
- **Self-attention (multi-head)** — query/key/value matmul, then a **softmax** over key positions. *This is the hard layer for verification — softmax is curved, attention is bilinear in inputs, and bounds blow up fast.*
- **MLP block** — two affine layers + GELU. *Sound bounds via CROWN-style relaxation; same family of techniques as Day-4 image-classifier verification.*
- **Output softmax** — converts logits to next-token probabilities. *The decision is the argmax (or a sample); the “robustness” question is on the **distribution**, not the argmax — a brand-new specification problem.*

The Explainer is a great **teaching prop**: every box you can click on is a layer some verifier needs to support. The honest answer to “what’s missing for *Let’s Verify ChatGPT*” is: **sound bounds for attention + softmax, at billions-of-parameters scale, against a non- ℓ_∞ specification on a discrete-token output**. (Shi et al., *Robustness Verification for Transformers*, ICLR 2020, is the standing formal anchor — and it stops well short of full LLMs.)

Adversarial perturbations → LLM jailbreaks: the same family

The image attacks earlier (Eykholt's stickers, our PGD demo) and "LLM jailbreaks" are the *same algorithm family* in different metric spaces.

	Image classifier (PGD)	LLM guardrail (GCG)
Substrate	continuous pixels in $[0, 1]^d$	discrete token sequence
Constraint	ℓ_∞ ball of radius ε	suffix of k tokens appended to a prompt
Objective	minimize true-class margin	maximize probability of a forbidden completion
Algorithm	gradient sign + project (Madry et al., ICLR 2018)	GCG : greedy coordinate descent on token logits (Zou et al., 2023 ; arXiv:2307.15043)
State of the art	falsifier; verifier exists (α, β -CROWN, NNV)	falsifier; no mature sound verifier

Demo: [day04/examples/nn/adversarial_demo.py](#) — FGSM (Goodfellow et al., ICLR 2015) and PGD on a small ReLU MLP. Watch each attack flip the prediction as ε grows; PGD is strictly stronger than FGSM at the same budget (the script asserts this as a sanity check).

The takeaway for the audience: **generating an attack is easy; proving none exists is the open problem.** That asymmetry is exactly the soundness $\leftrightarrow$ completeness gap from the whole week, and it is the reason this is still a research field.

Neural-network verification: the literature

Fact-checked entry points — the spine of the field:

- **Hardness & first solver** — Katz et al., *Reluplex*, CAV 2017 (exact ReLU robustness is NP-complete); modern successor **Marabou 2.0**, Wu et al., CAV 2024.
- **Family (a): bound propagation + B&B** — *CROWN* (Zhang et al., NeurIPS 2018) → α -*CROWN* (Xu et al., ICLR 2021) → β -*CROWN* (Wang et al., NeurIPS 2021) → *GCP-CROWN* (Zhang et al., NeurIPS 2022).
- **Family (b): reachability / abstract domains** — *A/2* (Gehr et al., IEEE S&P 2018); *DeepZ* (NeurIPS 2018) & *DeepPoly* (Singh et al., POPL 2019); **NNV star sets** (Tran et al., FM 2019), *ImageStar* (CAV 2020), *NNV 2.0* (Lopez et al., CAV 2023); *Verisig* for NN-controlled hybrid systems (Ivanov et al., HSCC 2019).
- **Books / surveys** — Albarghouthi, *Introduction to Neural Network Verification* (free at verifieddeeplearning.com); Liu et al., *Algorithms for Verifying Deep Neural Networks*, Found. & Trends in Optimization, 2021.
- **Hands-on tutorials** — our **AAAI'26 VNN-COMP tutorial** (slides + Google Colab notebooks): <https://vnn-comp.github.io/#aaai2026>; and the AAAI-2022 NN-verification tutorial (auto_LiRPA, α, β -CROWN Colabs) at neural-network-verification.com.
- **Frontier** — Shi et al., *Robustness Verification for Transformers*, ICLR 2020 — the formal anchor; full LLMs remain out of sound-verification reach.

Classical FM in production: AWS

AWS **Provable Security** runs formal methods at production scale:

- **s2n-TLS** — crypto kernels (HMAC, DRBG) verified with **SAW** against Cryptol specs (the L2 workflow), every commit.
- **s2n-bignum** — verified **assembly** (P-256/384/521, X25519, Ed25519, RSA, AES-XTS), each with a **HOL Light** proof, per commit.
- **Cedar** — authorization policy language **specified in Lean** (Day 3; originally Dafny).

Classical FM in production: everyone else

- **Semiconductors** — Cadence JasperGold, Synopsys VC Formal (Apple, Intel, AMD, ARM).
- **Verified systems** — seL4 (microkernel, Isabelle), CompCert (C compiler, Coq).
- **Avionics / space** — SCADE & Astrée (Airbus fly-by-wire), SPARK/Ada (Ariane 6).
- **Microsoft** — Dafny, Boogie, Z3 across Azure and Windows components.

One shape behind CBMC, SAW, and NN verification

Three very different Day-4 tools, one mental model — **compile the artifact to a formula, hand it to a solver:**

Tool	Artifact	Formula it builds	Solver
CBMC	C program + assertion	loops unrolled → SAT/SMT	SAT / Z3
SAW	C/LLVM vs Cryptol spec	symbolic execution → "impl = spec?"	SAT / SMT
α, β -CROWN / NNV	network + ℓ_∞ ball	bounds / reachable set → "bad output reachable?"	LP / MILP (linear / mixed-integer programming) / SMT

Each asks the solver the *same* question — **is a bad behavior satisfiable?** — and reads **UNSAT** as a proof. It is the Day-1 move (assert the negation) wearing three costumes.

Where formal methods meets AI

The convergence, both directions:

- **FM for AI** — verify neural networks (α, β -CROWN, NNV) and gate agent/LLM outputs (AWS Bedrock Automated Reasoning).
- **AI for FM** — LLMs draft *and repair* proofs, specs, invariants, and harnesses: retrieval-augmented proving (**LeanDojo / ReProver**, **Lean Copilot**), **AlphaProof** (IMO-medal level), DeepSeek-Prover, and **autoformalization** (natural language $\rightarrow$ Lean).
- **How we know it works — benchmarks:** **miniF2F / ProofNet / PutnamBench** (proving), **VERINA** (verifiable code-gen), **SV-COMP** (software), **VNN-COMP** (neural nets) — the field's honest scoreboards.
- The constant: a **trusted checker** (SMT kernel, Lean kernel) arbitrates. AI proposes; the kernel disposes — and a benchmark says by how much.

How do we know it works? Evaluating AI + FM

If AI proposes and a kernel disposes, *measuring* progress means counting what the kernel **accepts** — never what merely looks plausible.

- **pass@k** — give the model **k** tries; a problem counts as solved only if *some* attempt is **accepted by the checker** (Lean kernel, SMT proof, SV-COMP validator). No credit for plausible-looking text.
- **Benchmarks are the scoreboard:**

Benchmark	Domain	Measures
miniF2F / PutnamBench	competition & undergrad math	kernel-accepted proofs (pass@k)
SV-COMP	C software	correct verdict + machine-checked witness
VNN-COMP	neural networks	proved/disproved robustness; standardized ONNX + VNN-LIB

"Don't trust — verify." An LLM's Lean proof of a hard problem isn't read for plausibility; it is fed to the kernel. Accepted $\Rightarrow$ it counts. That is *how we know*.

The week in one table

Day	Tool	Question answered
1	Z3 + SMT-LIB	counterexample of length $\leq N$? — bounded refutation
2	nuXmv	counterexample ever? — full model checking
3	Lean 4	why no counterexample? — inductive proof
4	CBMC + Cryptol/SAW	does the source match the spec? — implementation-level

Same counter, **five encodings**, four kinds of guarantee.

L3 recap

- **Neural-network verification** (α, β -CROWN, NNV) verifies the AI systems themselves; VNN-COMP tracks the frontier.
- **Industry runs FM at scale** — AWS (SAW/Lean/Z3), chips (JasperGold), verified systems (seL4, CompCert).
- **FM $\leftrightarrow$ AI converge**, with a trusted checker always arbitrating.

Wrap + work

Day 4 — and the week — in one slide

- Programs are transition systems; **CBMC** verifies real C with Day-1's engine.
- **Cryptol** specs + **SAW** prove implementations equal their specs, over all inputs.
- The frontier: **verifying neural networks**; the reality: **FM in production everywhere**.
- The constant across four days: **AI proposes, the kernel disposes**.

What you can do now

You can:

1. **Encode** a system as a transition system.
2. **Specify** correctness in propositional, temporal, or higher-order logic.
3. **Discharge** it via SAT/SMT, model checking, or theorem proving.
4. **Connect** the spec to real code via CBMC and Cryptol/SAW.
5. **Drive** AI assistants in the loop, with the kernel as truth.

Where to go next — beyond this week

A 4-day intensive *samples*; here's the rest of the map, with courses that go deep (annotated list in the repo README):

- **Deductive verification, deeper** — we introduced contracts + loop invariants today; go further into **separation logic** (the heap, aliasing, concurrency) and full functional correctness with **Dafny**, **Verus** (verifies *Rust*), **Frama-C**, **Why3**, **Viper**. (CMU 15-414, ETH Program Verification.)
- **Program synthesis** — the dual of verification: **SyGuS**, **Rosette**. If generation is cheap, *synthesize-then-verify* is the natural pairing. (Berkeley 219C, OPLSS.)
- **Verified-stack tradition (Coq/Rocq, Isabelle)** — **CompCert** (C compiler), **seL4** (microkernel): the largest machine-checked artifacts. (DeepSpec, MIT FRAP.)
- **Probabilistic & hybrid model checking** — **PRISM**; nuXmv's infinite-state/IC3 and hybrid modes. (Oxford CAV.)
- **Protocol verification** — **Tamarin**, **ProVerif**, **CPSA** (**Cryptographic Protocol Shape Analyzer**, MITRE) for crypto *protocols* (vs. SAW's implementation proofs). (SRI SSFT.)
- **Evaluating AI + FM** — the scoreboards: **miniF2F** (proofs), **VNN-COMP** (NN), **SV-COMP** (C). The highest-value skill for the agentic era.

In-session exercise

Pick a track (see `assignments/day04.md`):

- **CBMC**: write a small C function + harness (saturating counter, queue, `my_abs`); verify a property; introduce a bug and capture the counterexample.
- **Cryptol/SAW**: write a small spec (4-bit cipher, parity, CRC-4) with two definitions; `:prove` they agree.

Each topic ships a `_starter` stub (which produces a counterexample) plus a worked solution — start from the starter and fix it until the verdict is clean.

Homework + survey (ungraded, for depth)

- Finish your chosen track to a clean verdict; capture one counterexample.
- **Survey paragraph** — pick one: latest VNN-COMP (what's in reach for α, β -CROWN?), AWS Provable Security s2n (what's continuously verified, with what stack?), or one thing in *your* research that is / isn't a fit for FM.

References for Day 4

- **Clarke, Kroening, Lerda.** *A Tool for Checking ANSI-C Programs*, TACAS 2004 (the original CBMC paper; lead/maintainer Daniel Kroening).
- **Lewis, Martin.** *Cryptol: High Assurance, Retargetable Crypto Development*, MILCOM 2003.
- **Galois** — Cryptol & SAW docs/tutorials; **AWS Provable Security** blog.
- **VNN-COMP** — <https://vnn-comp.github.io/>; **α, β -CROWN**; **NNV** (verivital). Hands-on **AAAI'26 VNN-COMP tutorial** (ours — slides + Google Colab notebooks): <https://vnn-comp.github.io/#aaai2026>.
- **NN-verification reading** — Katz et al. *Reluplex* (CAV 2017); Tran et al. *NNV / star sets* (FM 2019; ImageStar CAV 2020; NNV 2.0 CAV 2023); Albarghouthi, *Introduction to NN Verification* (free); Liu et al. survey (FnT Optimization 2021).
- **Frontier / next directions** — Johnson, *Is Neural Network Verification Useful and What Is Next?* (Allerton 2025, [hdi 2142/130315](#)); Serbinowska et al. *BehaVerify / neuro-symbolic behavior trees* ([verivital/behaverify](#)); Belcak et al. *Small Language Models are the Future of Agentic AI* (arXiv 2506.02153).
- **AWS Cedar** (Lean spec); **seL4**, **CompCert** (verified systems).

Full list + competitions table: repo [README.md](#).

Thank you

Four days, four pillars, one counter, five encodings.

Repo: github.com/ttj/fmaiv — all slides, examples, the Docker toolchain, and references.

Questions, and where to go next: the reading list and communities in the README.