Principles of Computer Systems (MIT 6.826 Fall 2020)
Course Notes

Prof. Nickolai Zeldovich

1 Course Overview

- **Focus:** Correctness of complex computer systems through principled specifications and verification techniques.
- **Prerequisites:** Exposure to systems (e.g., 6.033, 6.006, 6.828).
- **Key Questions:** What makes a system correct? How do we prove correct execution under concurrency, faults, and optimizations?

2 Motivation and Complexity

- Concurrency: multiple threads/machines introduce nondeterminism.
- Distribution: network failures, partial machine crashes.
- Faults: power failures, disk/memory errors (fail-stop vs Byzantine).
- Performance and optimizations add subtle bugs.
- Evolution and maintenance amplify complexity.

3 Fault Tolerance and Crash Safety

3.1 Crash Safety Challenges

- Disk writes are atomic at sector granularity but not multi-block.
- System may crash at any point, leaving partial updates.
- Disk controllers may reorder buffered writes.

3.2 Write-Ahead Logging (WAL)

Ensures atomic multi-block updates via a log region:

1. **Log writes:** Write new block contents to log area.
2. **Commit record:** Update header indicating pending transaction.
3. **Apply updates:** Copy logged blocks to data region.
4. **Cleanup:** Clear header to complete transaction.

3.3 Recovery Procedure

- On reboot, read header. If transaction pending, replay log to data region.
- Replay is idempotent: reapplying writes is safe.

3.4 Barriers and Ordering

- Barrier between log and commit to ensure log durability before header update.
- Barrier between commit and apply to ensure header persisted before data writes.
- Barrier before cleanup to ensure data writes complete.

3.5 Optimizations and Subtleties

- **Log Bypass Writes:** Non-atomic data writes bypass log with careful barrier.
- **Checksum Logging:** Replace barrier by checksum of log region in header.
- Combination of optimizations can introduce subtle bugs (e.g., ext4 metadata leakage).

4 Principled Verification

4.1 Testing vs Model Checking vs Verification

- Testing finds bugs but cannot prove absence.
- Model checking explores state-space (limited by state explosion).
- Formal verification: code + specification + proof ensures correctness.

4.2 Verification Workflow

- **Code:** System implementation (e.g., file system, protocol).
- **Specification:** Formal definition of correctness properties.
- **Proof:** Manual or automated arguments that code meets spec.
- Tools: Coq, Lean, SMT solvers (Z3, CVC).

4.3 Success Stories

- AWS service design verification (Amazon engineers).
- CompCert: Verified C compiler.
- Verified cryptography in Chrome and Firefox (e.g., Ed25519, assembly generation).

5 Course Logistics

- Weekly lectures + paper discussions (paper summaries required).
- Lab assignments in Coq Proof Assistant (Fault-tolerant storage, replicated systems).
- Participation: Submit questions and answers on readings.
- Grading: Labs, paper summaries, participation.

6 Lecture: Amazon Paper Discuss & Intro to Specifications

6.1 Breakout Room Activity

Students were split into groups of 3–4 to discuss:

> "The authors of the Amazon paper write formal specifications, yet do not use them to prove code correctness. What value do these specs bring to Amazon?"

After 5 minutes, groups shared key insights:

- **Design verification:** Specs expose design-level bugs early (e.g., logic loopholes or unintended behaviors), before any coding begins.
- **Evolution support:** With a spec in place, iterative changes can be validated against it, catching regressions without full reimplementation.
- **Documentation:** Formal specs serve as precise, unambiguous documentation—crucial in a large organization with many teams.
- **Forcing function:** Writing specs shifts focus from the "happy path" to all possible behaviors (safety/liveness), leading to more robust designs.

6.2 Prof. Lampson's Perspective on Specifications

1. **Modularity:** Decouple client code from implementation details.
2. **Insight:** Highlight *what* the system must do, abstracting away *how*.
3. **Proof of correctness:** (Later labs/papers) Show that implementation traces are a subset of spec traces via simulation arguments.
4. **Model checking:** Automate bug finding by exploring reachable traces against the spec.

6.3 TLA$^+$ Basics

- **Actions** are predicates on current and next state, e.g. $x' = x + 1$.
- **Non-determinism:** Combine actions with $\vee$; if multiple enabled, behavior is unpredictable.
- **Invariants:** Describe reachable states; crucial to reason about code only in valid states.

6.4 Spec Writing & Verification Workflow

1. Define *state variables* (keep this minimal to capture client-visible behavior).
2. Specify *operations* (transitions) in high-level notation (sets, comprehensions, nondeterministic choice).
3. (Optional) Perform *model checking* to catch bugs automatically.
4. For critical modules, derive *abstraction function* and perform simulation proofs:

$$c \xrightarrow{\text{code}} c' \quad \Longrightarrow \quad f(c) \xrightarrow{\text{spec}} f(c')$$

6.5 Key Takeaways

- Specs are *not* just for proofs—they drive design, documentation, and testing.
- Amazon's use of specs + model checking trades conclusive proofs for automation and speed.
- In this course, we'll explore both model checking (Amazon style) and full formal proofs (Coq labs, research papers).

7 Lecture: Specifications & Abstractions

7.1 Homework Check-In

- **Software Foundations exercises:**
  - Homeworks (Coq tutorials) straightforward once syntax is learned.
  - Readings (Chapter proofs) more notation-heavy—expect greater clarity when you apply them in labs.

- *Tip:* Pattern-match existing proofs, and use Piazza/office hours for syntax-level questions.

7.2 Why Specs & Abstractions?

- **Goal:** Reason about all possible executions of some code.
- **Spec → Proof → Confidence:**

$$\{\,\mathrm{Pre}(s)\,\}\; f\; \{\,\mathrm{Post}(s', r)\,\}$$

  where s, s' are program states, r the return value.
- **High-level vs. low-level views:**
  - *State-machine view* (Butler): global states → transitions, traces, invariants.
  - *Hoare logic view* (today): function calls take $s \to s'$, specs as pre/post predicates.

7.3 Hoare Logic Primer

- $\{P\}\ f\ \{Q\}$: if $P(s)$ holds, then f terminates in s' with $Q(s', r)$.
- **Partial correctness:** $\{P\}f\{Q\}$ says *if* f returns, Q holds; total correctness adds termination.
- **Sequencing rule:**

$$\frac{\{P\}\, x\, \{R\} \quad \{R\}\, y\, \{Q\}}{\{P\}\, x; y\, \{Q\}}$$

7.4 StaffDB Example

- **Code:**

```
def add(x):
  total := total + x
  count := count + 1

def average():
  require count > 0
  return total / count
```

- **Low-level specs (primitives):**

$$\{\,\top\,\}\ \mathit{read\_total}\ \{\, s' = s \wedge r = s.\mathit{total}\,\}, \quad \{\,\top\,\}\ \mathit{write\_total}(v)\ \{\, s'.\mathit{total} = v,\ s'.\mathit{count} = s.\mathit{count}\,\}$$

- **Composing via sequencing:** decompose $\mathit{add}(x)$ into $\mathit{read\_total};\ \mathit{write\_total}(t + x);\ \cdots$

7.5 Abstract-State Spec for StaffDB

- *Spec state:* $h \in \mathit{list}(\mathbb{N})$ = *history of inputs.*
- **High-level spec:**

$$\{\,\top\,\}\ \mathit{add}(x)\ \{\ h' = h[x]\ \}, \qquad \{\,|h| > 0\,\}\ \mathit{average}()\ \{\ r = \tfrac{\sum h}{|h|}\ \}$$

7.6 Abstraction Relation

$$R(s,h) \;\equiv\; s.total = \sum h \quad \wedge \quad s.count = |h|.$$

- $\{\, \exists h.\, R(s,h) \,\}\ f\ \{\, \exists h'.\, R(s',h') \wedge \Phi(h,h') \,\}$
- Layers proofs: once $\{R\}\, f\, \{\Phi, R\}$ holds, we can treat f as a single abstract step on h.

8 Discussion: Everest Paper

8.1 Breakout Group Reports

- **Core challenges noted:** confusion over the one-page "cryptographic game" description, the interplay of F*'s memory model, and the status of Everest vs. Fully-Verified Everest.
- **Key insight:** They embed each real implementation call under an "ideal" oracle (the "magic log") and prove the same high-level API spec holds whether you use the ideal model or the real cipher/MAC.

8.2 Magic-Log Model of Encryption

- *Ideal oracle:* upon `encrypt(key,p)`, return a fresh random `c`; record $\langle p, c\rangle$ in Log_{key}.
- *Decryption:* on `decrypt(key,c')`, look up $\langle p, c'\rangle \in Log_{\text{key}}$; return p if found, else $\perp$.
- *Security rationale:* without the key $\Rightarrow$ no access to Log, ciphertexts are uniform random.
- *Analogy:* one-time pad is a real-world instantiation: key = huge random pad, ciphertext = pad$\oplus$plaintext.

8.3 Nonce Usage

- **Nonce:** unique per-message, prevents replay—ensures identical plaintexts yield distinct ciphertexts.
- **Replay protection:** receiver tracks seen nonces; rejects duplicates.

8.4 Everest Project Overview

- **Goal:** drop-in replacement for OpenSSL/TLS stack with machine-checked correctness.
- **Stack layers:**
  1. *Crypto primitives* in F*/Low* $\rightarrow$ C (AES, ChaCha, Poly1305, etc.).
  2. *Verified assembly* (via VEIL) for performance-critical loops.
  3. *Parser/Serializer* (EverParse) for ASN.1, X.509, DER.
  4. *TLS Handshake & Record Protocol* in F*.
  5. *HTTPS Interface* compatible with OpenSSL API.
- **Deployment:** extract Low*$\rightarrow$C; compile with CompCert or GCC; "drop-in" for existing servers and clients.

8.5 Threat Model & Attacks

- **Heartbleed:** example of buffer-overflow in parsing; allowed arbitrary memory disclosure.
- **Man-in-the-Middle (MITM):** exploit certificate misuse or protocol bugs to intercept/decrypt.
- **Side-channels:** timing-dependent branches on secret key; mitigated by constant-time coding and memory-access patterns.

- **Certificate authentication:** X.509 chains, root CAs, NSS trust anchors; primary source of real-world misconfigurations.

8.6 Open Questions

- How well will Everest resist large-scale deployment attacks (phishing/MITM) versus opportunistic bugs?
- Can its $F^\star$ proofs scale to cover the full OpenSSL API surface without compatibility regressions?
- What performance-cost trade-offs remain after enforcing constant-time and fully verified parsers?

8.7 Limitations of Abstraction Functions

- *Insufficient state:* cannot record past executions (no "history") or anticipate future choices (no "prophecy").
- *Augmentations:*
  - **History variables** log every visited state/transition.
  - **Prophecy variables** predict which future transition will occur.
- *Completeness:* any implementation–spec trace inclusion can be witnessed by combining abstraction, history, and prophecy variables.

8.8 Trace Inclusion: Code vs. Spec

Definition A program **implements** its spec if every externally visible trace of the code is also allowed by the spec.

Safety "If the code returns a result, it satisfies the spec."

Liveness "The code eventually returns a result (i.e. terminates)."

8.9 Sequential Example: Sorting

- **Spec:** relation on input/output arrays requiring the output to be a sorted permutation.
- **Code:** deterministic sort (e.g. `quicksort`) picks one allowed output.
- *External trace:* only initial and final arrays.
- *Internal steps:* pivot choices and swaps—hidden from the spec.

8.10 Simulation via Abstraction Functions

- An *abstraction function* $f : \text{CodeState} \to \text{SpecState}$ must satisfy:
  1. $\forall t_0$ initial: $f(t_0)$ is initial in the spec.
  2. If $t \xrightarrow{\pi} t'$, then $f(t) \xrightarrow{\pi} f(t')$.
- By induction, Traces(code) $\subseteq$ Traces(spec).
- A *state invariant* restricts attention to reachable states.

8.11 Example: Write-Back Cache

- **Spec:** memory $m : \texttt{Addr} \to \texttt{Val}$, operations READ/WRITE.
- **Code:** adds a cache $c : \texttt{Addr} \rightharpoonup \texttt{Val}$ plus main memory m.

- *Abstraction:* overlay $\mathsf{mem}(a) = c(a)$ if defined, else $m(a)$.
- *Invariant:* $|\text{dom}(c)| = C_{\mathsf{size}}$ is preserved by LOAD/FLUSH.

8.12 Extra Spec State & History Variables

- **Spec:** `DB` stores a list of inputs to compute mean/variance.
- **Code:** maintains only $(n, sum, sum2)$.
- *Solution:* add a history variable h (the full list) to code; relate $(n, sum, sum2)$ to $\text{fold}(h)$ via an invariant.

8.13 Abstraction Relations

- Generalize f to a relation $R \subseteq \text{Code} \times \text{Spec}$.
- If $(t, s) \in R$ and $t \xrightarrow{\pi} t'$, then $\exists\, s'$ with $s \xrightarrow{\pi} s'$ and $(t', s') \in R$.
- Supports many-to-one and one-to-many state mappings.

8.14 Internal Transitions

- Code or spec may take unobservable steps (*internal*).
- Simulation: a code step labeled π must match a sequence of spec steps whose visible projection is π.

8.15 Prophecy Variables

- Required when the spec makes a *premature choice*—e.g. dropping messages at crash time or agreement at `allow`.
- A prophecy variable p is chosen up front to predict which future branch will fire.
- Formal rules ensure prophecy does not disable real code steps and preserves the external trace.

8.16 Limitations of CompCert's Correctness Specification

- **Liveness**
  - Distinguishes *terminate* vs. *diverge*, but cannot decide termination (undecidable).
  - No guarantees that a non-faulting program will *eventually* produce output.
- **Performance**
  - No complexity or latency bounds: an efficient C algorithm may compile to a much slower binary.
  - CompCert itself may take unbounded time or memory on pathological inputs.
- **Memory Safety & Undefined Behavior**
  - Behaviors after UB (e.g. buffer overflows, integer overflow, null-pointer dereference) are unconstrained.
  - Only *safe* (UB-free) C programs are in scope; no protection if the source invokes UB.
- **Security Properties**
  - No guarantees on confidentiality or integrity (e.g. side-channel resistance, secret erasure).
  - Optimizations may expose secrets or enable timing attacks.
- **Application-Level Correctness**

- Preserves C semantics, but does not verify that the *application logic* is correct or meets its spec.

8.17 Static Analysis: A Lightweight Alternative

- **What is static analysis?**
  - A "lightweight cousin" of full formal verification: no global proofs, but automatic, scalable checks.
  - Universally adopted in industry to catch bugs early, without running the program.
  - Active research—new tools and techniques emerge constantly.
- **Case studies: Google `FindBugs` vs. Facebook `Infer`**
  - Both tools check for *partial* specifications (common bug patterns, API-misuse, null dereferences), not full program correctness.
  - Co-designed with real dev workflows—tool authors work closely with engineers to choose which properties to check.
- **Spec vs. developer goals**
  - *Full verification* demands a complete spec proof; static analysis targets *universal invariants* (no null-pointer use, no unchecked user input, etc.).
  - Emphasis on *actionable* warnings:
    * Low "effective" false-positive rate (Google): a warning is *not* false if a dev fixes it.
    * Low missed-bug rate (Facebook): focus on catching real in-the-wild defects (crashes, security, data races).
  - Feedback loop trust: compile-time review-time batch dashboards. Early, in-context alerts build trust and drive fixes.
- **Scalability via locality & compositionality**
  - Most checks are *intra-procedural*: look at a few lines or one function—fast, low overhead.
  - *Inter-procedural* bugs (null returned deep in call chains, unsanitized user input) require summaries:
    * `Infer` automatically infers per-function "mini-specs" to scale whole-program dataflow race detection.
  - Incremental, parallel analyses: each function can be analyzed independently, then recomposed.
- **Key takeaways**
  - Static analysis succeeds when it solves concrete developer pain points—fast feedback, low noise, clear fixes.
  - Tools must be integrated into IDEs or code-review (compile-time ideal) to minimize context-switch cost.
  - Co–design with engineers, monitor actionable fix rates, and tune analyses (precision vs. recall) to real workloads.

8.18 SybilFS: Specifying and Testing POSIX File Systems

- **Motivation:**
  - Real-world file systems (ext3, HFS+, etc.) follow informal POSIX "man-page" specs.
  - Goal: a *precise*, executable spec to drive exhaustive tests and uncover subtle bugs.

- Impact: influenced POSIX editors to tighten ambiguities.

- **Key challenge—non-determinism:**
  - POSIX leaves many behaviors *unspecified* (e.g. error-code ordering, bytes-returned by `read()`, directory-entry order).
  - Concurrency in `readdir()`—interleaved creates/deletes yield many possible valid traces.
  - Must capture "all implementations" under one spec.

- **SybilFS approach:**
  - *Lem DSL* for spec:
    - Define abstract OS state (process table, open-file map, directory contents).
    - Label transitions: `call(pid, op, args)`, `return(pid, result)`, plus for reordering concurrency.
    - Non-deterministic choice: Lem's "|||" to enumerate all allowed outcomes.
  - *Workload generator:*
    - Automatically explore syscall sequences to drive corner-case behaviors.
    - No oracle needed—SybilFS "oracle" is spec membership check.
  - *Online checking:*
    - Track *set* of possible spec states matching the observed trace so far.
    - After each system call / return label, prune spec states whose transition label observed label.
    - Empty match set implementation–spec divergence (bug!).

- **Directory iteration model:**
  - Maintain per-`opendir()` "*must*" and "*may*" sets for entries present throughout vs. those concurrently created/deleted.
  - Guarantees:
    - "Must" entries always returned.
    - "May" entries may or may not appear, in any order, possibly interleaved.

- **Results and takeaways:**
  - SybilFS found both spec ambiguities and real file-system bugs in Linux, BSD, macOS.
  - Precise, executable specs power stronger black-box testing than ad hoc workloads.
  - Non-determinism modeling + aggressive pruning keeps state-space manageable.

8.19 Separation Logic: Foundations and Modular Reasoning

- **Why Separation Logic?**
  - Tackles pointer-aliasing by *separating conjunction* ($P * Q$): asserts P, Q hold on disjoint heap fragments.
  - Enables concise *local* specs and proofs for heap-manipulating programs.
  - Scales to concurrency (ownership transfer) and to large codebases (Facebook Infer).

- **Core Assertions:**
  - `x   v`: "cell$x$ contains v."
  - $P * Q$: P holds on one part of the heap, Q on a disjoint part.
  - `emp`: the heap is empty.
  - *Entailment* $P \vdash Q$: whenever P holds, so does Q.

- **Inductive Predicates:**
  - `list(x)` or `tree(x)`: describe linked structures by recursion.
  - Example:

$$list(x)\ \begin{cases} x = null: \ emp, \\ x \neq null: \ \exists d, z.\ x \mapsto (d, z) \ * \ list(z). \end{cases}$$

  - Extensions can track *contents*: `list_addr(x,L)` pairs shape with stored data L.
- **Specifying Procedures:**
  - *Hoare triple*: $\{P\}\ C\ \{Q\}$ means "if P holds, then after C, Q holds."
  - Example `prepend(x,a)`:

$$\{\ list(x)\ \}\ \texttt{new}(r, a, x)\ \{\ list(r)\ *\ list(x)\ \}.$$

  - Garbage-collecting a tree:

$$\{\ tree(t)\ \}\ \texttt{delete\_tree}(t)\ \{\ emp\ \}.$$

- **Local Reasoning—Frame Rule:**

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \quad \text{(if } C \text{ does not touch } R\text{).}$$

  - Modify only the footprint of P, leave R intact.
  - Yields highly *modular* proofs—reason about one heap fragment at a time.
- **Mechanization with Iris:**
  - Uses *weakest precondition* $\mathsf{WP}\, e\{Q\}$ instead of triples.
  - Spatial context lists separating-heap hypotheses; proof tactics mutate them in-place.
  - Recursive calls handled by assuming WP spec holds on subcalls.

8.20 Verified File System (FSCQ)

- **Why Verify a File System?**
  - *Critical infrastructure*: all real systems persist data onto a filesystem.
  - *Stable spec, buggy implementations*: POSIX-style behavior rarely changes, yet crashes and subtle bugs still occur.
  - *Crash safety*: power loss or kernel panic may strike at any point—must prove no on-disk corruption.
  - *Asynchronous disk I/O*: controller buffers writes unpredictably; crash may lose or reorder pending writes.
- **FSCQ Artifact**
  - `Gallina` implementation of a simple FS + logging layer.
  - `Coq` proofs over a *CrashCore* logic: functional correctness *and* crash-recovery guarantees.
  - `Extraction` to Haskell, compiled as a `FUSE` filesystem—live mountable on Linux.
- **Abstract Disk Model**
  - Addresses $\mapsto$ *lists* of values: pending writes collect in order.
  - `read(a)` returns the *last* entry in list; does not drop older entries.
  - `write(a,v)` appends v to the list at a.

- `sync` "flushes": collapse each list to its last element only.
- `crash` + `recover`: for each a, nondeterministically pick one entry from its list and discard the rest.

- **Logging Layer as Two Synchronous Disks**
  - *Active disk*: collect `log_write` into a transient, synchronous "logical" disk.
  - *Committed disk*: last-committed state, also synchronous.
  - `commit`: atomically copy entire Active→Committed.
  - `recover` after crash: reset Active←Committed.
  - Yields *all-or-nothing* semantics: pending writes never leak unless `commit` finishes.
- **Crash-Core Logic Extension**
  - Each operation spec carries (1) a *post-condition* for normal return, and (2) a *crash-condition* for mid-execution failure.
  - `write(a,v)` crash-condition: either before or after write—but never "partial" sector update.
  - Global proof of `recover()`:
    - Show every step's crash-condition implies the `recover` pre-condition.
    - *Idempotence*: `recover`'s crash-condition equals its pre-condition, so repeated crashes during recovery remain safe.

8.21 Concurrency and x86 TSO

- **Motivation**: real-world CPUs expose *weak* memory behaviors—hardware optimizations (caches, store-buffers, OOO, speculation) break the intuitive SC view.
- **Sequential Consistency (SC)**
  - All loads and stores appear as *atomic* steps on a single, shared memory.
  - Programmers can interleave per-thread steps but never observe "out-of-thin-air" reorderings (e.g. $\langle 0, 0\rangle$ in the classic two-store/two-load example is impossible).
  - Clean abstraction, but too expensive for high performance.
- **Weak Memory Toys:**
  - *Store-buffering*: stores go into a per-core buffer before hitting memory.
  - *Loads* may read from the buffer or from main memory atomically.
  - *Background/driven flushes* and explicit `MFENCE` let buffered writes propagate.
  - *Speculation & out-of-order execution* further reorder effects unless fenced.
- **TSO Abstract Machine**:
  - One "hardware thread" per SMT lane; each has its own store buffer.
  - `Load(a)` atomically returns the newest pending or committed value.
  - `Store(a,v)` enqueues $(a \mapsto v)$ in the local buffer only.
  - `Fence/MFENCE` forces buffer → memory flush before continuing.
  - `LOCK-prefixed` instructions perform their R/W plus buffer flush atomically.
- **Litmus-Test Examples**:
  - Two-store/two-load "MP" test admits $(0, 0)$ under TSO (stores sit in buffers).
  - "Independent-Reads-Of-Independent-Writes" (IRIW) only fails if two readers share a buffer—TSO forbids it with per-thread buffers.
- **OS vs. Hardware Threads**:

- OS context switch must *flush* a process's buffer (or treat it as empty) when descheduling, to preserve TSO at the user level.
- Kernel/user boundary (`IRET`) on x86 implicitly acts as a fence.

- **Axiomatic Specs (Intel/AMD)**:
  - Informal English "rules" + forbidden litmus tests—hard to cover all cases, and sometimes inconsistent with actual chips.
  - Contrast: TSO paper provides a crisp, formal *abstract machine* ideal for both reasoning and teaching.
- **Key Takeaway**: *TSO's simple cartoon (per-thread store buffers + fences) captures exactly what x86 offers—enabling correct low-level concurrency without drowning in cache/coherence/speculation details.*

8.22 Finding Concurrency Bugs (`TSVD`)

- **Why concurrency bugs are hard:**
  - *Coverage explosion*: #threads×#interleavings grows combinatorially.
  - *Poor reproducibility*: a rare schedule may trigger a crash once but then never recur.
- **Testing vs. static analysis**:
  - *Static analyzers* (lock-set, thread-sanitizer) must infer cross-call contexts; often too imprecise or expensive.
  - *Dynamic testing* runs real interleavings—but must both define a "bug" and drive schedules to expose it.
- **Bug definition in TSV-D:**
  - Each `TSV` (thread-safety-violation) is a *conflicting* read/write or write/write on an API-specified data structure.
  - API authors annotate each `TType` with a "read set" (methods that only observe) and "write set" (methods that mutate).
  - Violation ≡ two methods from these sets run *concurrently*.
- **No false positives (by design):**
  - "Bug" ≡ observed violation of their TSV contract (e.g. concurrent `Map.Add()` vs. `Map.TryGet()`).
  - Benign data races (e.g. counters) are excluded because only library-declared APIs are monitored.
- **Schedule-driving via delays ("traps")**
  - *Trap points*: on each API call, record (object, op).
  - Inject `Sleep()` before a trapped operation to amplify "near-miss" races.
  - If another thread's conflicting API runs during the sleep, report a TSV.
- **Heuristics to avoid wasted delays:**
  - *Near-miss detection*: only trap when two ops on the same object occurred within Tms in recent execution.
  - *Delay-inference*: if delaying one op also delays its partner by $\approx T$, infer they're synchronized (e.g. via a lock) and stop trapping there.
- **Strengths & limits:**
  - Very low user effort: drop-in .NET tool finds real bugs in standard collections.

– Low "false alarm" rate on API-annotated data structures—over half of reported violations were fixed by developers.
– Misses bugs in un-annotated or custom APIs; may over-delay in hot paths.

8.23 Lab 3 Recitation: Crash-Safe Log

- **Motivation:**
  1. *Atomicity via write-ahead log*: append all entries, then "commit" by updating a header.
  2. *Crash safety*: on crash, recovery must see either the old log or the fully appended new log—no torn writes.
  3. *API-design practice*: define `get`, `append`, `reset` purely by postconditions, then implement from first principles.
- **Logging API spec (in `log_api.file`):**
  - `state: list<block>` (unbounded).
  - `get()`: returns entire log (no state change).
  - `append(xs)`:
    - May succeed (return `true` and extend log) or fail (return `false`, no change).
    - *Crash-safety*: if crash occurs during append, final log is either old or old++xs.
  - `reset()`: clears log (atomic or no-op on crash).
- **Underlying disk API:**
  - Fixed-size array of blocks, atomic single-block read/write.
  - We build multi-block `append` plus crash-safe commit.
- **Crash-safe design (header + payload):**
  - On disk, block 0 stores `n = current log length`.
  - Blocks $1..\,n$ hold valid log entries; blocks $> n$ are garbage.
  - `get()`: read header n, then read blocks $1..n$.
  - `append(xs)`:
    1. Write entries of `xs` into blocks $n + 1 \ldots n + |xs|$.
    2. Finally write updated length $n + |xs|$ into header.
  - Crashing before header write leaves header at old n (old log); crashing after gives new log.
  - `reset()`: single atomic write of `0` to header.
- **Pseudocode sketch:** [basicstyle=] `get(): n ← disk.read(0) return foldr (i from 1 to n) (++ [disk.read(i)]) []`
  `append(xs): n ← disk.read(0) if n + |xs| + 1 > DISK`$_SIZEthenreturnfalsefori, xinenumerate(xs) : disk.write(n + 1 + i, x)disk.write(0, n + |xs|)returntrue$
  `reset(): disk.write(0, 0)`
- **Verification strategy:**
  - *Abstraction relation* between disk array and logical log:

$$(\exists n).\ 0 \leq n \leq \mathit{DISK\_SIZE} - 1 \ \wedge\ \big(\forall i < n.\ \mathit{disk}[i+1] = \mathit{log}[i]\big).$$

  - Prove `get`, `append`, `reset` satisfy their specs under crash semantics.
  - Use loop-combinator lemmas (`for_range`, `for_each`) with tailored loop invariants.
  - Automate common disk-update rewrites with `auto_rewrite_with_upd`; discharge arithmetic side-conditions via `lia`.

8.24 Proving Concurrency Correct

- **Today's Goals:**
  1. Understand how to prove a concurrent implementation refines its specification.
  2. See what is *easy* vs. *hard* in concurrency proofs.
  3. Learn how to build "large" atomic actions from smaller ones.
- **Two Views of "Spec vs. Code":**

  State-Machine View: – Global states s, atomic "steps" $A(s, s')$; define a *trace* or *behavior* as a sequence of states.
  - A *spec* = set of allowed traces; code $\subseteq$ spec $\iff$ code's visible traces are in spec.
  - Proof by *invariants*: find $I(s)$ such that

  $$I(s_0) \wedge A(s, s') \implies I(s'),$$

  then I holds forever.

  Language/Command View: – Primitives: pure expressions, $x := e$. Composition via ;, if, while, $\|$, etc.
  - Semantics via *weakest preconditions* (or Hoare triples): $wp(c, Q)$.
  - Proofs by *wp-calculus* or triples rather than global invariants.
- **Threads in State Machines:**
  - Each thread t has its own program counter pc_t in a large state.
  - "Next" relation is $\bigvee_t A_t(s, s')$, where A_t fires only if pc_t matches.
  - Invariant must hold after *every* thread's step.
- **Refinement/Data Abstraction:**
  - A mapping $m : S_{code} \to S_{spec}$. Lift to traces pointwise.
  - Code implements spec *under* m if $\forall \tau_c.\, trace_c(\tau_c) \implies trace_s(m(\tau_c))$.
  - Proved by showing $init_c \implies m^{-1}(init_s)$ and $next_c \implies m^{-1}(next_s)$.
- **Atomic Actions:**
  - *Hardware-provided*: e.g. load/store of one word, test&set.
  - *By composition*: group $a; b$ into one atomic action if b "commutes" with every action in other threads:

  $$a; b \subseteq b; a \quad \text{(as sets of traces).}$$
- **Commutativity Cases:**
  1. *Disjoint variables:* actions touch different vars.
  2. *Producer–Consumer:* only communication via put/get on a buffer.
  3. *Locking:* a holds lock ℓ, b also requires ℓ—so b cannot interleave.
  4. *Abstraction:* replace a complex sequence by an atomic "black-box" once proven correct.
- **Mutex Acquire/Release:**
  - `acquire(m)`: atomically wait for $m = \mathit{free}$, then set $m = \mathit{self}$.
  - `release(m)`: if $m = \mathit{self}$, set $m = \mathit{free}$ (else havoc).
  - *Two-phase locking:* hold *all* needed locks before touching shared data, then release at end.
- **Simulation Proof Sketch** of atomicity for $a; b$:

  - Show $\{ab\} \subseteq \{ba\}$ by case analysis on whether the interfering action c interleaves before or after.
  - Use a relational invariant linking "a done" vs. "not yet done" in the other schedule.

- **PlusCal Example:**
  - A simple `spinlock` using atomic `TestAndSet`:

```
while TestAndSet(m) = 1 do skip od;
```

    but *bad* on single-CPU: no one else can release.
  - A *realistic* multi-processor lock (Lamport's bakery / spinlock variants):
    * Processes numbered $1 \dots N$, each with label-guarded steps.
    * Carefully placed assertions (`Assert`) to encode invariants at key labels.
    * Proof obligation: for each thread PC ℓ, every other thread's step preserves all asserts.

- **Key Takeaways:**
  - Always try to fit your concurrency protocol into *disjoint*, *producer–consumer*, or *locking* patterns.
  - If you stray into "hard" concurrency (no commuting discipline), you must do a full correctness proof or risk elusive bugs.
  - PlusCal/TLA+ give a *state-machine* style with invariants; the *language* (wp) style scales to code-level but needs external mapping to hardware.

8.25 Reading *Armada*: Mechanized Concurrency Proofs

- **Admin: Lab4 Options**
  - Default: prove *replicated disk* with crash safety in our current framework.
  - Alternatives (notify instructors early): explore Daphne, Iris, VST, or free-form project.

- **Why *Armada*?**
  - Machine-checked proofs of *fine-grained* concurrent code, *without locks.*
  - Realistic x86-TSO memory model, not just sequentially consistent.
  - Illustrates *state-machine* reasoning at scale and automation techniques.

- **Proof Foundations:**

  Invariants –global state predicates preserved by every step.
  Abstraction Relations –relate concrete state-machine transitions to high-level spec.
  Mover/Reduction –classify each code step as *right-/left-/both-/non-mover.*
    - Right-movers can be delayed past others; left-movers can be advanced earlier.
    - Sequence of (rights) $n\,(lefts)$ compresses into one atomic action.

- **Armada Pipeline:**
  1. Write both *spec* and *code* in the same Armada language (plus nondet. $*$, `choose`).
  2. *Translator* generates an explicit state machine: states = full memory+PCs; steps = individual atomic transitions.
  3. Developer annotates *reduction strategy* (regions of right/left movers, fences, TSO-elimination).
  4. *Proof generator* emits Daphne proof scripts showing:

$$Traces_{\text{code}} \subseteq Traces_{\text{spec}}.$$

5. Daphne (with Z3 backend) discharges thousands of small commuting and invariant-preservation lemmas.

- **Key Mechanization Trick: "Sigma"**
  - Each transition is a *deterministic* function $\mathit{next}(s, \sigma) \rightarrow s'$ by packaging all nondet. choices (malloc result, thread-ID, branch $*$) into a `step` record σ.
  - Commutativity proof becomes a simple *equality* check:

$$\mathit{next}(\,\mathit{next}(s, \sigma_i), \sigma_j) \;=\; \mathit{next}(\,\mathit{next}(s, \sigma_j), \sigma_i).$$

- **Automation vs. Manual Proof:**
  - State-machine approach: many small, uniform obligations $\rightarrow$ amenable to SMT automation.
  - Language-based (Iris, separation logic): fewer but more *creative* invariants, harder to auto-solve.
- **Next Lecture:** Compare with *Iris*'s *language-level* concurrency logic and verify how its proof style and tooling differ from the state-machine + mover approach.

8.26 Reading the Iris Blog Post

- **Motivation:**
  - *Iris* embeds concurrency reasoning as a *program logic*, not a state machine.
  - Blog post walks through a toy "bank" example to expose *lock invariants*, *ghost state*, and `iProp` proof mode.
  - Goals today:
    1. Explain *lock invariants* via Go/Rust idioms and Iris's formalization.
    2. Introduce *ghost variables* (fractional permissions, update, splitting).
    3. Read and interpret a typical Iris proof obligation (`WP`— Iris's weakest-precondition goal).
- **From Functional to Concurrent Imperative:**
  - Original spec: `transfer(bank,b,n) = new_bank` in pure functional style.
  - Single-threaded imperative: pointers & separation logic to prove that `transfer` preserves total sum.
  - Naïve concurrent spec fails (interleaved transfers break "start = b" assumption).
  - We want each account's transfer to be atomic *and* composable with other independent transfers.
- **Lock Invariants:**
  - *Go*: convention "// m protects b1,b2"—only a comment, unenforced.
  - *Rust*: `Mutex<Balances>` ties data to the lock type; scope-based unlock.
  - *Iris*: associate each lock with an *invariant* P s.t.

$$\{\, \texttt{lock}(l)\}\; \texttt{acquire}\; l\; \{P\} \quad \text{and} \quad \{P\}\; \texttt{release}\; l\; \{\, \texttt{lock}(l)\}$$

  - Inside P one may bundle both *points-to* assertions and *pure facts* (e.g. balances sum to 0).
- **Ghost Variables & Fractional Permissions:**

- *Ghost var* $\mapsto^q v$ holds that "ghost γ has value v with fraction $q \in (0,1]$."
- Rules:
  1. *Allocation*: introduce fresh $\gamma \mapsto^1 v$.
  2. *Split / Combine*: $\gamma \mapsto^1 v \Leftrightarrow \gamma \mapsto^q v * \gamma \mapsto^{1-q} v$.
  3. *Persistence*: two fragments $\gamma \mapsto^{q_1} v * \gamma \mapsto^{q_2} v \implies v$ equal.
  4. *Update*: owning $\gamma \mapsto^1 v$ lets you change to any v'.

- **Bank Example in Iris:**
  - Two accounts `b1`, `b2` each protected by its own lock invariant:

$$\exists b, v.\ (\texttt{b1} \mapsto b * \gamma_1 \mapsto^{\frac{1}{2}} v * \Box\,(b = v)) \quad * \quad (\text{same for } b2, \gamma_2)$$

  - *Global* invariant: $\gamma_1 \mapsto^{\frac{1}{2}} v_1 * \gamma_2 \mapsto^{\frac{1}{2}} v_2 * (v_1 + v_2 = 0)$.
  - `transfer(n)` proof outline:
    1. `acquire b1; acquire b2`—get both lock invariants.
    2. Update physical pointers: `*b1 -= n; *b2 += n`.
    3. `open` both invariants to gain full ghost ownership, update γ_1, γ_2, then `close`.
    4. `release b2; release b1`—re-establish each P, including $b = v$ and sum 0.
  - `check_consistency()` acquires both locks, `open` global invariant, checks $v_1 + v_2 = 0$, then `close`&`release`.

- **Reading Iris `WP` Goals:**

  Persistent context facts duplicable across threads (`is_lock`, *invariants*).
  Spatial context exclusive resources (*points-to*, *ghost perms*).
  ***WP* triple** mixes Iris proof-mode steps (‘`iIntros`, `iDestruct`, `iCombine`, ...‘) with standard separation logic.

- **Beyond the Example:**
  - *Custom ghost state*: monotonic counters, authoritative sums, spatial region algebras.
  - *Atomicity specs*: you can prove `transfer` itself is logically atomic (*vs.* merely safe).
  - *RustBelt*: semantic type-soundness of `Vec<T>`— unsafe implementation, safe API guaranteed by Iris.

- **Next Time:** We'll continue exploring Iris by mechanizing a small concurrent stack and seeing how *higher-order* invariants and *fancy updates* extend these techniques.

8.27 IronFleet: Verified Distributed Systems

- **Why Distributed Systems?**
  - Concurrency across machines
  - High communication costs (e.g. cache misses or network latency)
  - Partial failures—must remain available despite node crashes

- **Five-Layer Verification Architecture**
  1. *Abstract Spec* (à la Lamport): global "God's-eye" view with a visibility relation.
  2. *Protocol Level*: hosts execute `ph_next` actions atomically; communicate by message sends/receives; prove a global invariant in TLA/PlusCal style.
  3. *Host Code*: each host action is pure sequential code; prove it refines the protocol's atomic action via reduction (movers).

4. *Network Model*: UDP-style packets; maintain a ghost-journal of sends/receives to reason about message flow.
5. *Composition*: combine host+network to get the full distributed system; then compose refinements up to the abstract spec.

- **Key Proof Technique: Reduction & Movers**
  - *Process actions*: commute with everything (both movers).
  - *Receive actions*: right-movers (can be delayed).
  - *Send actions*: left-movers (can be advanced).
  - Any host-action matching R–⟶T–⟶L pattern is atomic by commuting out other hosts' steps.
- **Example1: Distributed Lock**
  - Spec: sequence of holders; each `Acquire` appends the holder's ID.
  - Impl: hosts send "grant" and "ack" packets with a counter; invariant tracks last `Acquire` packet to reconstruct holder sequence.
- **Example2: Replicated State Machine (Paxos RSM)**
  - Spec: behave like a single deterministic machine on a command stream.
  - Protocol: classic Paxos rounds—propose, accept/quorum, learn; must preserve *quorum-intersection* to ensure agreement.
  - Optimizations: batching, leader election (view changes), state transfer (snapshot), reply caching.
  - Liveness under timed fairness assumptions (beyond pure asynchrony).
- **Example3: Rebalancing Key–Value Store**
  - Data partitioned by key range; may "move" a range by sending in-flight split packets.
  - Invariant: every key is either owned by exactly one host or in a pending transfer packet.
  - Reliable transmission and ordering layered over UDP.
- **Pragmatic Considerations**
  - Trust assumptions: spec, compiler/runtime, OS, hardware.
  - Verified libraries for containers, marshalling, data-structure invariants.
  - Ghost state for unbounded history (network journal).
  - Automation trade-offs: powerful SMT vs. careful annotations; modular proofs over large codebases.
  - 4× code overhead; needs expert proof engineers; constraints on code shape for automation.

8.28 Ivy/I4: Automated Protocol Invariant Discovery

- **Context: Protocol Verification**
  - A sub-problem of distributed-system verification, distinct from implementation correctness.
  - Separates "does the protocol work?" from "can I write correct code to implement it?"
  - Fits into IronFleet's five-layer stack at the *protocol-level* (between host code and global spec).
- **Why Protocols Are Hard**
  1. *Unreliable Networks*: messages may be lost, delayed, duplicated, reordered.

2. *Node Failures*: crashes vs. network partitions are indistinguishable; recovering state is tricky.
3. *Dynamic Membership / Byzantine Faults*: (beyond I4's scope) arbitrary misbehavior complicates consensus.

- **Ivy: SMT-Backed Protocol Checker**
  - User writes `state` relations (e.g. `semaphore(s)`, `link(c,s)`).
  - Defines `init` predicate and `action` transitions.
  - Safety spec = predicate on reachable states (e.g. no two clients hold same lock).
  - Ivy encodes: [label=()]
  - *Init* $\Rightarrow$ *Invariant*,
  - *Action preserves Invariant* as Z3 queries $\Rightarrow$ fully automated safety proof.
- **Inductiveness vs. Safety**
  - *Safety bubble*: all states satisfying spec predicate.
  - *Reachable bubble*: all states reachable by stepping from `init`.
  - Spec predicate *may not* be inductive (closed under transitions).
  - Need a *stronger* inductive invariant (blue bubble) satisfying: [label=()]
  - init $\subseteq I$,
  - $\forall s \in I,\ s \to s' \implies s' \in I$,
  - $I \subseteq$ spec.
- **I4: Automatic Invariant Inference**
  - Leverages a bounded-model checker (AVR) to exhaustively explore a *small* instance (e.g. 1server, 2clients).
  - AVR synthesizes a compact formula characterizing *all* reachable states in that instance.
  - I4 lifts/generalizes this formula to *unbounded* parameters to propose a global inductive invariant.
  - Feed back into Ivy; if too weak, increase small-model bounds and repeat.
- **Demo: Lock Service**
  - Simple client–server lock protocol in Ivy.
  - Initial spec fails inductiveness test (must forbid "server still holds lock" when client thinks it does).
  - Strengthen invariant (add `link(c,s)` $\implies$ $\neg$`semaphore(s)`).
  - Ivy checks *init* and every *action* automatically in seconds.
  - Add "client-to-client transfer" action; Ivy again verifies safety with no manual proof.
- **Trade-Offs & Applicability**
  - Extremely easy to verify safety of unbounded protocols (no Coq/Tac scripting).
  - Relies on: decidable fragment of logic, small-model generalization.
  - *Deadlock* is a liveness—not safety—concern (Ivy supports some liveness via cycle-finding).
  - Open question: can small-model invariant inference extend to richer concurrency proofs (e.g. Iris/Armada)?

8.29 Verifying Software-Defined Networks

- **Motivation**
  - *Critical infrastructure*: Every distributed system relies on IP/Ethernet forwarding.
  - *Narrow, well-scoped spec*: Packet-in/packet-out behavior vs. arbitrary stateful services.

- *High complexity*: Distributed control (switches, failures, reconfiguration) hides bugs.

- **Traditional Networks**
  - Each switch independently stores *(config, routing state)*.
  - Switches run distributed protocols (e.g. OSPF, BGP) to build forwarding tables.
  - *Challenges*: Inconsistent configs, complex failure recovery, per-switch debugging.

- **Software-Defined Networks (SDN)**
  - *Centralized controller*: Single global "brain" programs all switches.
  - *Data-plane switches*: Fast path uses locally cached *flow tables*.
  - *Benefits*: Simplified policy, hot-swap hardware, unified vendor API.
  - *Risks*: Controller single point of failure; reconfiguration must preserve connectivity.

- **NetCore/Featherweight OpenFlow**
  - *NetCore DSL*:
    * `match cond` on packet headers
    * `modify` primitive header fields (TTL, IP)
    * `action` selects output ports
    * `union/restrict` to combine rules
  - *Flow table IR*: Ordered list of {`match,modify,action`} entries, resolved by priority.
  - *Featherweight OpenFlow*: Controller–switch protocol
    * `PacketIn` $\rightarrow$ controller if no match
    * `Add/DeleteFlow` from controller to switch
    * `Barrier` to enforce order

- **Correctness via Certified Compilation**
  - *Compile-time*: NetCore $\xrightarrow{\text{certified compiler}}$ controller binary + runtime
  - *Theorem*: Controller+runtime $\simeq$ NetCore spec
  - *Trace inclusion*:
    * *Implementation* traces $\subseteq$ *Spec* traces (safety)
    * *Spec* traces $\subseteq$ *Implementation* traces (bisimulation $\approx$ liveness)

- **Limitations & Outlook**
  - *Static configurations*: Paper models one fixed NetCore program—no dynamic updates.
  - *Controller reliability*: Single-host performance and fault-tolerance not addressed.
  - *Higher-level policies*: End-to-end liveness (e.g. "all flows are logged") must be layered atop NetCore.

8.30 Empirical Study of "Verified" Distributed Systems

- **Paper goal:**
  - *Ask*: Do complex, machine-checked DS actually eliminate bugs?
  - *Approach*: Audit three systems (IronFleet, Verdi, Chapar) for real faults.

- **Why bugs persist:**
  - *Spec gaps*: What the proof assumes vs. real API behavior.
  - *Shim errors*: Unverified "glue" layers (OS, network, I/O) violate axioms.
  - *Tooling faults*: Build scripts or provers skip or ignore proof failures.

- **Finding bugs:**

- *Fuzzing shims*: Inject resource errors, partial I/O, packet loss/duplication.
- *Cross-checking*: Compare against alternate implementations or hand-written tests.
- *Manual audit*: Inspect build logs, proofs, spec comments for mismatches.

- **Representative faults**

  IronFleet
  - *Tooling*: Build script ignores Z3 exit signals; proof errors go unnoticed.
  - *Spec*: "Exactly-once" duplicate filtering not guaranteed by spec.

  Verdi
  - *Shim*: TCP receive may yield partial or no messages; file partial writes crash on replay.
  - *Tooling*: Deep recursion in extracted OCaml overflowed stack (no liveness guarantee).

  Chapar
  - *Shim*: UDP axioms omitted packet loss/duplication; custom marshal API left stale bytes.
  - *Spec*: Causal-consistency invariants broken by unchecked network behavior.

- **Lessons & best practices**
  - *Lean, precise spec*: Drive proofs by writing and verifying small example apps atop your spec.
  - *Integral proof workflows*: Always require explicit "success" outputs, not just absence of errors.
  - *Harden shims*: Fuzz and test every OS / network primitive; prefer narrow, verified APIs.
  - *Layered verification*: Push boundaries sensibly—too large → complexity, too small → unsound assumptions.
  - *Operational checks*: Combine formal guarantees with staged rollouts, runtime monitoring, and alarmed fallbacks.

8.31 Komodo: Minimal-Hardware Enclaves via Verified Monitor

- **Motivation:**
  - Intel SGX provides secure enclaves in hardware—but is complex, hard to extend.
  - *Goal*: Recreate enclave isolation & attestation with minimal hardware, pushing policy into software.
- **Security background:**
  - *Isolation* trades off with *sharing*—must authenticate who may access which resource.
  - *AuthN/AuthZ* via `guard` mediating requests against system policy.
  - *Attestation*: map concrete channels (e.g. crypto pipes) to high-level principals via "speaks-for" chains.
  - *Threats*: hostile OS, side-channels, induced faults, denial of service.
- **Enclave architecture:**
  - *Host* (OS/VMM) is untrusted; software monitor and enclave code must enforce security.
  - *Monitor*: tiny "baby hypervisor" mediates transitions (`SMC`, exceptions, interrupts) between:
    * Normal world (untrusted OS)
    * Enclave world (trusted code)
  - *Hardware support* (if only against software threats):
    * Protected RAM region (OS can't touch).
    * Secure control-transfer instructions in CPU.

    * Root key for attestation; RNG for crypto.

- **Attestation protocol:**
  1. `attest(key)`: monitor returns $\mathrm{MAC}_{HK}(ms, key)$, binding enclave measurement ms to signing key.
  2. `verify(key,ms,tag)`: check MAC under hardware root key HK "key speaks for ms".
  3. Chain trust: hardware key monitor enclave.

- **Formal verification:**
  - *Spec*: 12 Monitor calls plus enter/exit semantics; enforces:
    * *Confidentiality*: public outputs depend only on public inputs.
    * *Integrity*: trusted outputs depend only on trusted inputs.
  - *Model*: ARM machine model in "Veil" pseudo-assembly; opaque oracles resolve non-determinism.
  - *Proof*:
    * Verify each Monitor transition implements spec (Dafny+Z3).
    * Non-interference (relational refinement) over world-switch boundaries.

- **Key takeaways:**
  - Small, verified monitor proves enclave isolation, attestation—avoids SGX microcode complexity.
  - Even tiny code bases harbor corner-case bugs—verification catches subtle "page A=pageB" errors.
  - Strong spec + minimal TCB + verified toolchain yields high assurance with modest hardware.

8.32 Non-Interference and Confidentiality in CertiKOS

- **Integrity vs. Confidentiality**
  - *Integrity* (functional correctness) ensures "no corruption" of state.
  - *Confidentiality* means "no unauthorized disclosure" of secrets.
  - Confidentiality is much harder: must prevent any leakage, not just wrong answers.

- **Example: Two-Block Disk**
  - Block0 holds userA's data, Block1 holds B's.
  - Naïve rule "B never reads 0" still allows many leaks (out-of-bounds reads, metadata APIs, remappers).
  - Any non-determinism in spec or implementation can be exploited to distinguish A's secret.

- **Non-Interference as Two-Safety**
  - *One-trace safety*: "no single bad trace."
  - *Two-trace safety* (non-interference): for any two initial states indistinguishable to B, all B's observations along both executions must remain identical.
  - B's entire visible behavior—reads, outputs, syscalls—must be independent of A's secret.

- **Observation Functions**
  - $Obs_{\text{spec}}(p, s)$: what principal p is allowed to see in abstract state s.
  - $Obs_{\text{code}}(p, c)$: what p actually observes at the implementation level.
  - Must satisfy $Obs_{\text{code}}(p, c) \subseteq Obs_{\text{spec}}(p, s)$ whenever c implements s.

- **Proof Outline**
  1. *Spec-level determinism*: every abstract step from $s \to s'$ preserves Obs_{spec}.
  2. *Lowering*: if two spec states are indistinguishable, their code states remain indistinguishable under Obs_{code}.
  3. By induction on steps, B's final observations cannot distinguish A's secret.
- **Challenges and Corner Cases**
  - *Specification leaks*: forgetting to include page-table layout or PID allocation in Obs_{spec} can break proof.
  - *Implementation leaks*: exposing extra channels (e.g. "used-blocks" API) not modeled in Obs_{code}.
  - *Concurrency nondeterminism*: context switches break the two-trace alignment → solved by "local semantics," collapsing other threads into a single yield step.
- **Takeaways**
  - True confidentiality requires reasoning about *pairs of executions* (two-safety).
  - Complete determinism (in spec & code) simplifies proofs but is often impractical.
  - Designing precise observation functions is crucial: they define both allowed spec observables and actual code leakage.
  - Practical non-interference for OS kernels (like CertiKOS) must handle VM mappings, syscalls (fork/PID), and concurrency carefully.

8.33 What Formal Proofs Give—and Don't—for Security

- **Paper context**
  - Authors: Toby Murray (sCL4 microkernel), security and verification experts.
  - Genre: philosophical "meta-paper" on the gap between *proved* theorems and *real-world* security.
  - Goal: set realistic expectations for using proofs in security projects.
- **Why proofs seem ideal for security**
  - Security is a *negative goal*: "no attacker can ever break in," so every corner case matters.
  - Formal proofs force you to *consider all cases* and eliminate human oversight.
  - A concise, correct *specification*—if achievable—yields machine-checked confidence.
- **Why proofs alone may fall short**
  1. *Mis-modeled reality*: CPU models often omit nondeterminism, timing, undocumented registers, or rarified instructions.
  2. *Incomplete threat model*: hardware bugs (e.g. Rowhammer), side-channels, SMM/JTAG debug paths, physical tampering.
  3. *Uncaptured APIs*: e.g. PID allocators, "used-blocks" queries, speculative features.
  4. *Specification vs. implementation drift*: theorem may not say what you think, or be hard to interpret (weeks to grok seL4's statement!).
- **Code changes and "Venn diagram" of edits**
  - P: changes needed to make the proof go through.
  - A: changes needed for real-world security.
  - $P \cap A$: *ideal*—you only change what both demand.
  - $P \setminus A$: proof-overhead edits (e.g. off-by-one tweaks, proof-friendly refactorings).

  - $A \setminus P$: attacks your proof missed (e.g. Rowhammer bitflips, timing leaks).
  - $\exists\, V \subseteq (P \setminus A)$: *worse edits* that actually weaken security.

- **Value of proofs, despite limits**
  1. *Qualified guarantees*: "system is secure *if* these precise (and extractable) assumptions hold."
  2. *Structured exploration*: writing down state, spec, abstraction & proof uncovers bugs & clarifies design.

- **Defense in depth**
  - Even with a *proved* memory-safe engine, you still layer ASLR, canaries, sandboxing, etc.
  - Backup measures mitigate the inevitable *threat-model drift* encountered in practice.

8.34 Automated, "Push-Button" Verification with Rosette & Servo

- **Motivation:** Eliminate
  - Low-level memory/overflow bugs (buffer overrun, div0, UB).
  - Logical errors (missing sanity checks, path-specific flaws).
  - *Design* bugs (API flaws that break isolation or leak secrets).

- **Illustrative UB in C:**
  - Multiply two 16bit `uint16_t` via `c = (uint32_t)(a*b)`
  - `-O0` yields correct $a * b$, `-O2` triggers signed-overflow UB and returns "wrong" value.
  - GCC exploits "signed-overflow is UB" to optimize away.

  $\Rightarrow$ even a one-line "innocent" routine can go wrong under real compilers.

- **"Push-Button" Verification Stack:**

  Rosette $\rightarrow$ **Servo** $\rightarrow$ *your verifier* $\rightarrow$ SMT solver

  - *Rosette:*
    - Embeds your interpreter or DSL in a symbolic language.
    - Lifts *concrete* interpreter into *symbolic* evaluator.
    - Provides *knobs* (symbolic reflection, custom providers) to tune encodings.
  - *Servo:*
    - Framework atop Rosette for low-level code (RISC-V, LLVM IR, BPF).
    - Builds "no-proof" verifiers: spec + implementation $\xrightarrow{\text{Servo}}$ SMT.
  - *Jitterbug:*
    - A Servo-based JIT-compiler verifier for LinuxBPF:
      - Found and fixed real bugs in the upstream kernel.
      - Shipped in Linux since March 2024.

- **Symbolic vs. Bounded Encoding:**
  - *Pure symbolic execution:* forks at every branch, merges later $\rightarrow$ *path-explosion*.
  - *Bounded model-checking:* one-step "merge" after each instruction $\rightarrow$ huge symbolic terms.
  - *Rosette's hybrid:* uses *type-guided merges* to keep encoding size polynomial & precise.

- **Profiling & Tuning "Magic Box":**
  - *Symbolic profiler* spots expensive eval sites (e.g. symbolic PC in an interpreter).

- *Custom provider* (e.g. `split-pc`): force concrete cases on PC, collapse paths early.
- Iteratively "repair" your spec/interpreter until verification finishes.

- **Retrofitting Classic Verifiers:**
  - Ported seL4-style security monitors (CERTiKOS, Nova, etc.) to RISC-V + Servo.
  - Proved each system-call lemma (e.g. `yield`, `alloc`, `exit`) separately so that SMT can handle it.
  - Turned days of manual Coq proofs into ∼weeks of Servo setup per API.
- **Practical Impact:**
  - Verified new BPF "JIT" compiler and plugged it into Linux kernel.
  - Uncovered real bugs in both Linux core and ARM support libraries.
  - Demonstrated "push-button" verification can enter production—low manual-proof overhead.

8.35 Why Not Proofs? Engineering for Reliability

- **The "Software Crisis" (1960s–'90s):** Formal methods promised to tame exploding complexity, but industrial uptake was limited.
- **High-Reliability Case Studies:**
  - *Therac-25 (1985):* Sloppy UI+ concurrency bugs in radiation machine software → patient overdoses.
  - *Ariane5 (1996):* Unhandled FP-to-integer overflow in Ada spec caused dual guidance-computer failure → self-destruct.
  - *Telephone Exchanges (1970s–'90s):* Carrier-grade "six-9s" availability achieved with rigorous engineering, not proofs.
- **Economics of Reliability:**
  - Only mission-critical (avionics, banking, cloud infra) can justify the cost of exhaustive proofs.
  - Most software (desktop apps, web services) tolerates occasional bugs—"approximate" vs. "precise" software.
- **Tony Hoare's Recipe for Quality (1996):**
  1. *Rigorous Design& Review:* Inspect and cure specification flaws before coding.
  2. *Testing as QA:* Use tests to *drive* specs, detect faults, and feed back into design—*not* to "test in" quality.
  3. *Continuous Debugging:* Fix problems immediately in development and production (DevOps loop).
  4. *Over-Engineering & Fault Isolation:* Fail fast, restart components, isolate modules; accept redundancy.
  5. *Informal Math*: Leverage discrete-math ideas (invariants, pre/postconditions) in everyday specs.
- **When to Turn to Formal Methods:**
  - Concurrency and failures—rare, adversarial interleavings that evade testing.
  - Use lightweight *modeling* (TLA+/PlusCal+model-checking) to *design-verify* distributed protocols (e.g. Amazon S3).
  - Full machine-checked proofs reserved for small kernels or crypto stacks with huge consequences.

- **Other Key Lessons:**
  - *DevOps&Agile:* Developers operate their own code, enabling rapid feedback and regression control.
  - *Component Reuse & Moore's Law:* Off-the-shelf databases, languages, and GPUs tolerated by vast compute headroom.
  - *Technical Debt Awareness:* Regularly repay code "debt" before it blocks future feature delivery.
  - *Procurement and Partnership:* Cooperative customer–vendor relationships crucial—antagonism leads to failure.