

Software Foundations of Security & Privacy

Digest Notes

Mohamed Waiel Shikfa

These notes are a condensed companion to the official course lecture notes. Each lecture is summarized as: a goal, one-page map, core definitions, key proof patterns/recipes, common pitfalls, and a short “check yourself” set. Full rule lists and semantic definitions are collected in the appendices and referenced from the relevant lecture.

Lecture 2 (Jan 11, 2026): Propositional Logic and Proof

Goal: Use sequent calculus to (1) precisely define validity of formulas/sequents and (2) build/check formal derivations; understand soundness/completeness/decidability via invertibility + termination.

One-page map

- Objects: propositional formulas F built from variables and connectives; sequents $\Gamma \vdash \Delta$.
- Semantics: $\Gamma \vdash \Delta$ is valid iff “all Γ true implies some Δ true”.
- Proof system: right rules decompose goals; left rules decompose assumptions; identity closes branches.
- Meta-properties: soundness (derivable \Rightarrow valid), completeness (valid \Rightarrow derivable), decidability (procedure to decide validity).

Core definitions

- Definition. Formula: built from p using $\wedge, \vee, \rightarrow, \neg, \top, \perp$.
- Definition. Valid formula: true under every truth assignment.
- Definition. Satisfiable formula: true under some truth assignment.
- Definition. Sequent $\Gamma \vdash \Delta$: assumptions Γ and goals Δ (multi-succedent, disjunctive).
- Definition. Valid sequent: if all formulas in Γ are true then at least one formula in Δ is true.
- Definition. Derivation: tree built bottom-up using inference rules; leaves typically close by identity.

Key rules / theorems (high level)

- Key idea. Right/Left rules eliminate a top-level connective from a goal/assumption.
- Key idea. Disjunction needs multi-succedent sequents for completeness.
- Theorem (informal). Soundness: if $\Gamma \vdash \Delta$ is derivable then it is valid.
- Theorem (informal). Completeness + decidability: all rules are invertible and reductive, so proof search terminates and succeeds iff sequent is valid.

Proof-search recipe (sequent calculus)

- Step 1: Apply invertible rules bottom-up whenever possible (decompose connectives).
- Step 2: Stop at variable-only leaves $p_1, \dots, p_n \vdash q_1, \dots, q_m$.
- Step 3: Leaf closes iff some $p_i = q_j$ (identity). Otherwise it is a countermodel seed.

Pitfalls

- Common pitfall. Confusing formula implication $F \rightarrow G$ with sequent entailment $F \vdash G$: they coincide in validity but are different syntactic objects.
- Common pitfall. Using a two-rule disjunction right-introduction ($\vee R_1 / \vee R_2$) breaks completeness for classical truth-table validity. In general, avoid applying more than one rule at a time.

Check yourself

1. Check yourself. Prove $p \vee (p \rightarrow q)$ is valid using sequent calculus.
2. Check yourself. Given a failed leaf $p, q \vdash r$, what truth assignment witnesses invalidity?

Reference: Full propositional sequent-calculus rules are in Appendix [A](#).

Lecture 3 (Jan 13, 2026): Dynamic Logic

Goal: Extend sequent reasoning to talk about programs; express safety-style properties as postconditions; derive rules for program constructs (if/assignment/sequence/while).

One-page map

- New syntax: programs α (assign/seq/if/while) and modalities $[\alpha]Q$ (box) and $\langle\alpha\rangle Q$ (diamond).
- Intuition: $[\alpha]Q$ means “every terminating run of α ends in a state satisfying Q ”.
- Theme: Rules should reduce properties of compound programs to properties of smaller programs.

Core definitions

- Definition. Trace: (possibly infinite) sequence of states/events during execution.
- Definition. Safety property: “nothing bad happens” (prefix-closed).
- Definition. Liveness property: “something good eventually happens”.
- Definition. Box/diamond: $[\alpha]Q$ (all terminating runs end in Q) and $\langle\alpha\rangle Q$ (some terminating run ends in Q). Since our language has no nondeterminism, $\langle\alpha\rangle Q$ is equivalent to “ α terminates and $[\alpha]Q$ ”.

Key rules (as “recipes”)

- Key idea. If: split on the guard.
- Key idea. Sequential Composition: rewrite $[\alpha; \beta]Q$ as $[\alpha](\beta]Q)$.
- Key idea. Assignment: do not assume $x = e$ with the same x on both sides of the state change; introduce fresh x' (post-state value) and substitute.
- Key idea. While: unfolding is sound but not reductive; loop invariants are the scalable rule.

Assignment gotcha (why freshness matters)

- Common pitfall. The tempting rule “assume $x = e$ after $x := e$ ” is unsound because it confuses the value of x before and after the assignment.
- Lecture example: with the unsound rule you could (wrongly) justify $x = 2 \vdash [x := 1] x = 3$ by reducing it to the premise $x = 2, x = 1 \vdash x = 3$, which is valid only because the antecedents are contradictory.
- Key idea. Fix (as in the notes): introduce a fresh post-state variable x' and replace the postcondition $Q(x)$ with $Q(x')$. Rule shape:
$$\Gamma \vdash [x := e]Q(x), \Delta \text{ reduces to } \Gamma, x' = e \vdash Q(x'), \Delta \text{ (with } x' \text{ fresh).}$$

Loop-invariant recipe

- Pick an invariant J and prove these sequents (turnstile, not implication):
 - Init: $\Gamma \vdash J$
 - Preserved: $J, P \vdash [\alpha]J$
 - Post: $J, \neg P \vdash Q$
- Important: Γ (facts true only initially) is intentionally dropped from the preserved/post obligations.

Check yourself

1. Check yourself. Explain why $[\alpha]Q$ is vacuously true when α does not terminate. What property is this capturing?
2. Check yourself. For the swap program (using + and -), what must the precondition include to avoid undefined behavior in bounded-integer languages?

Reference: Full dynamic-logic language summary is in Appendix [B](#), and full rule summary is in Appendix [C](#).

Lecture 4 (Jan 20, 2026): Semantics of Programs

Goal: Pin down the meaning of expressions, programs, and formulas so rule soundness proofs become “expand definitions and chase the quantifiers”.

One-page map

- State: $\omega(x)$ is the value of variable x .
- Expression meaning: $\omega[e] \in \mathbb{Z}$.
- Program meaning: relation $\omega[\alpha]\nu$ between pre/post states.
- Formula meaning: $\omega \models P$.
- Reusable move: prove equivalences (e.g. $[\alpha; \beta]Q \leftrightarrow [\alpha](\beta]Q)$) and use them as rewrite rules.

Core clauses (keep only the high-yield ones)

- Definition. Assignment updates state: $\nu = \omega[x \mapsto \omega[e]]$.
- Definition. Sequence composes via an intermediate state: $\exists \mu$.
- Definition. While termination is “there exists an n iterations” (bounded-iteration semantics).

Reusable proof pattern: “unwind definitions”

- To show $P \leftrightarrow Q$ is valid: fix an arbitrary state ω , assume $\omega \models P$, expand definitions until you can derive $\omega \models Q$ (and vice versa).
- Key idea. This justifies using valid equivalences as both left- and right-rules (sound + invertible).

Pitfalls

- Common pitfall. Quantification over program states is subtle: naive “substitute a constant for x ” can fail when x is read across many loop states; semantics uses state update $\omega[x \mapsto c]$.

Check yourself

1. Check yourself. Prove (informally, by definitions) $\models [\alpha; \beta]Q \leftrightarrow [\alpha](\beta]Q)$.
2. Check yourself. Give a simple program α and postcondition Q where $\omega \models [\alpha]Q$ holds because α does not terminate.

Reference: Full semantics (expressions, programs, formulas) are summarized in Appendix D.

Lecture 5 (Jan 25, 2026): Proving Safety

Goal: Extend dynamic logic to account for unsafe execution (undefined-behavior-style), and learn proof patterns for showing programs avoid unsafe commands (often via invariants/guards).

One-page map

- Three outcomes of running a program: (i) a post-state ν , (ii) no post-state because it diverges, or (iii) no post-state because it goes unsafe.
- Semantic hook: introduce the unsafe judgment $\omega[\alpha]\not\models$.
- Safety as a DL formula: to show α is safe under precondition P , prove $P \rightarrow [\alpha]\top$.
- Engineering move: sandboxing = instrument unsafe commands with guards so the result is provably safe.

Core definitions

- Definition. Unsafe execution: $\omega[\alpha]\not\models$ means executing α starting in state ω performs an unsafe operation.
- Definition. Safe program (at ω): not unsafe, i.e. not $\omega[\alpha]\not\models$ (it may still diverge).
- Definition. Indeterminate behavior: execution is safe, but some expression results are unspecified; we keep expressions safe and push “badness” into commands.
- Definition. Updated box semantics (partial correctness + safety):

$$\omega \models [\alpha]Q \text{ iff } \left(\forall \nu. (\omega[\alpha]\nu \Rightarrow \nu \models Q) \right) \wedge \text{not } (\omega[\alpha]\not\models).$$

Key rules / recipes

- Key idea. To prove safety: aim for $P \rightarrow [\alpha]\top$ and then rely on rules that force you to prove the operation-specific safety condition.
- Key idea. Unsafe command example (division-as-command): $x := \mathbf{divide} e_1 e_2$ is unsafe when e_2 evaluates to 0.
- Key idea. Sequent rule shape for unsafe commands: one premise proves the safety side-condition, another carries the (possibly indeterminate) result into a fresh post-state variable.

Two “control” commands: assert vs test

- Assertion: **assert** P is unsafe when P is false (like “crash = unsafe”).
Key idea. Axiom: $[\mathbf{assert} P]Q \leftrightarrow (P \wedge Q)$.
- Guard/test: **test** P aborts safely when P is false (no poststate, but not unsafe).
Key idea. Axiom: $[\mathbf{test} P]Q \leftrightarrow (P \rightarrow Q)$.

Sandboxing recipe

- Transform: before any potentially unsafe command c , insert a guard **test** $\text{Safe}(c)$.
- Key idea. Why it works: after **test** $\text{Safe}(c)$, you can use the guard postcondition to discharge the safety premise required by the unsafe-command rule.

Pitfalls

- Common pitfall. Nontermination is not the same as unsafe: $[\alpha]Q$ can still hold vacuously when α diverges, but it must be false if α is unsafe.
- Common pitfall. Mixing up **assert** P and **test** P : the former is unsafe on failure; the latter aborts safely on failure.

Check yourself

1. Check yourself. Why is $[\text{test } \perp]Q$ valid for every Q under the updated box semantics?
2. Check yourself. What formula do you prove to show “division by e_2 is safe” inside a proof of $P \rightarrow [\alpha]\top$?

Lecture 6 (Jan 27, 2026): Memory Safety

Goal: Add a (bounded) memory model, define unsafe out-of-bounds reads/writes, and learn proof patterns for memory safety (often invariants with bounds + optional sandboxing via guards).

One-page map

- Memory model: a distinguished variable M denotes a heap/array $H : \mathbb{Z} \rightarrow \mathbb{Z}$.
- Bounded addresses: legal indices satisfy $0 \leq i < U$; out-of-bounds access is unsafe.
- New commands: read $x := M[e]$ and write $M[e_1] := e_2$ (unsafe if the index is out of bounds).
- Reasoning support: use theory-of-arrays style terms **read** $M e$ and **write** $M e_1 e_2$ in formulas (expressions remain safe but may be indeterminate).

Core definitions

- Definition. Safe read: $x := M[e]$ is safe iff $0 \leq e < U$.
- Definition. Safe write: $M[e_1] := e_2$ is safe iff $0 \leq e_1 < U$.
- Definition. State shape: lowercase variables map to integers; uppercase variables like M map to heaps/arrays.

Key rules (as recipes)

- Key idea. Read rule forces a bounds proof and introduces a fresh post-state variable:
 $\Gamma \vdash [x := M[e]]Q(x), \Delta$ reduces to $\Gamma \vdash 0 \leq e < U, \Delta$ and $\Gamma, x' = \mathbf{read} M e \vdash Q(x'), \Delta$.
- Key idea. Write is analogous with fresh M' :
 $\Gamma \vdash [M[e_1] := e_2]Q(M), \Delta$ reduces to $\Gamma \vdash 0 \leq e_1 < U, \Delta$ and $\Gamma, M' = \mathbf{write} M e_1 e_2 \vdash Q(M'), \Delta$.
- Key idea. Invariant pattern: to keep writes safe in loops, invariants usually need both a lower bound and the global U bound (e.g. $0 \leq i \leq n < U$).

Tiny example (invariant needs bounds)

- Program: $i := 0; \mathbf{while} (i < n) \{M[i] := i; i := i + 1\}$.
- Key idea. To prove $0 \leq n < U \rightarrow [\dots] \top$, a good invariant is $0 \leq i \leq n < U$ (not just $i \leq n$).

Sandboxing memory access

- Key idea. Instrument reads/writes:
 - replace $x := M[e]$ by **test** $0 \leq e < U ; x := M[e]$;
 - replace $M[e_1] := e_2$ by **test** $0 \leq e_1 < U ; M[e_1] := e_2$.
- Key idea. Result: the sandboxed program is provably safe (often with invariant \top), but may abort.

Pitfalls

- Common pitfall. For loop preservation, you drop the initial-context assumptions Γ ; invariants must carry the needed bounds themselves.
- Common pitfall. After a write you must reason about the new heap M' (freshness), not the old M .

Check yourself

1. Check yourself. Why does the invariant $i \leq n$ fail to prove safety for $M[i] := i$ inside the loop body? Name the missing facts.
2. Check yourself. Write the sandboxed version of $x := M[y + 1]$ and explain (briefly) what property it guarantees.

Appendices

A Propositional sequent calculus (full rule list)

Sequents: $\Gamma \vdash \Delta$ where Γ, Δ are (multi)sets of formulas. Validity: all Γ true implies some Δ true.

Identity

$$\frac{}{\Gamma, F \vdash F, \Delta} \text{id}$$

Logical connectives

True	$\frac{}{\Gamma \vdash \top, \Delta} \top R$	$\frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta} \top L$
False	$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \perp, \Delta} \perp R$	$\frac{}{\Gamma, \perp \vdash \Delta} \perp L$
Negation	$\frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta} \neg R$	$\frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta} \neg L$
Conjunction	$\frac{\Gamma \vdash F, \Delta \quad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta} \wedge R$	$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta} \wedge L$
Disjunction	$\frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta} \vee R$	$\frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \vee G \vdash \Delta} \vee L$
Implication	$\frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \rightarrow G, \Delta} \rightarrow R$	$\frac{\Gamma \vdash F, \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \rightarrow G \vdash \Delta} \rightarrow L$
NOR (\downarrow)	$\frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma \vdash F \downarrow G, \Delta} \downarrow R$	$\frac{\Gamma \vdash F, G, \Delta}{\Gamma, F \downarrow G \vdash \Delta} \downarrow L$

B Dynamic logic: Language summary (incl. safety + memory)

Int vars	x, y, z
Memory var	M
Constants	$c ::= \dots, -1, 0, 1, \dots$
Expressions	$e ::= c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid \dots \mid e_1/e_2 \mid \mathbf{read} M e \mid \mathbf{write} M e_1 e_2$
Programs	$\alpha, \beta ::= x := e \mid \mathbf{skip} \mid \alpha ; \beta \mid \mathbf{if} P \mathbf{then} \alpha \mathbf{else} \beta \mid \mathbf{while} P \alpha \mid \mathbf{test} P \mid \mathbf{assert} P \mid x := \mathbf{divide} e_1 e_2 \mid x := M[e] \mid M[e_1] := e_2$
Formulas	$P, Q ::= e_1 \leq e_2 \mid e_1 = e_2 \mid \top \mid \perp \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \neg P \mid [\alpha]Q \mid \langle \alpha \rangle Q$

C Dynamic logic: rule summary (box modality)

Reading: $[\alpha]Q$ is partial correctness (all terminating runs end in Q).

$$\begin{array}{c}
 \frac{\Gamma, P \vdash [\alpha]Q, \Delta \quad \Gamma, \neg P \vdash [\beta]Q, \Delta}{\Gamma \vdash [\text{if } P \text{ then } \alpha \text{ else } \beta]Q, \Delta} [\text{if}]R \\
 \frac{\Gamma, x' = e \vdash Q(x'), \Delta}{\Gamma \vdash [x := e]Q(x), \Delta} [=]R^{x'} \\
 \frac{\Gamma \vdash [\alpha](\beta]Q), \Delta}{\Gamma \vdash [\alpha ; \beta]Q, \Delta} [:]R \\
 \frac{\Gamma, P \vdash [\alpha](\text{while } P \alpha]Q), \Delta \quad \Gamma, \neg P \vdash Q, \Delta}{\Gamma \vdash [\text{while } P \alpha]Q, \Delta} [\text{unfold}]R \\
 \frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\text{while}_J P \alpha]Q, \Delta} [\text{while}]R
 \end{array}$$

D Semantics cheat sheet

States: total maps ω that map int variables to \mathbb{Z} and memory variables (like M) to heaps $H : \mathbb{Z} \rightarrow \mathbb{Z}$.

Unsafe execution

$$\omega[\alpha]\notin$$

means executing α from ω performs an unsafe operation (so it has no poststate).

Expressions

$$\omega[c] = c \quad \omega[x] = \omega(x) \quad \omega[e_1 + e_2] = \omega[e_1] + \omega[e_2]$$

(and similarly for other operators).

State update

$$(\omega[x \mapsto c])(x) = c \quad (\omega[x \mapsto c])(y) = \omega(y) \text{ for } y \neq x$$

Programs as relations Write $\omega[\alpha]\nu$ for “executing α can take prestate ω to poststate ν ”.

$$\begin{aligned}
 \omega[x := e]\nu &\text{ iff } \nu = \omega[x \mapsto \omega[e]] \\
 \omega[\alpha ; \beta]\nu &\text{ iff } \exists \mu. \omega[\alpha]\mu \wedge \mu[\beta]\nu \\
 \omega[\text{if } P \text{ then } \alpha \text{ else } \beta]\nu &\text{ iff } (\omega \models P \wedge \omega[\alpha]\nu) \text{ or } (\omega \not\models P \wedge \omega[\beta]\nu) \\
 \omega[\text{while } P \alpha]\nu &\text{ iff } \omega[\text{while } P \alpha]^n\nu \text{ for some } n \in \mathbb{N}
 \end{aligned}$$

with the bounded-iteration clauses:

$$\begin{aligned}
 \omega[\text{while } P \alpha]^0\nu &\text{ iff } (\omega \not\models P \wedge \omega = \nu) \\
 \omega[\text{while } P \alpha]^{n+1}\nu &\text{ iff } (\omega \models P \wedge \exists \mu. \omega[\alpha]\mu \wedge \mu[\text{while } P \alpha]^n\nu)
 \end{aligned}$$

Formulas

$$\begin{aligned}
 \omega \models [\alpha]Q &\text{ iff } (\forall \nu. (\omega[\alpha]\nu \Rightarrow \nu \models Q)) \wedge \text{not } (\omega[\alpha]\notin) \\
 \omega \models \langle \alpha \rangle Q &\text{ iff } \exists \nu. (\omega[\alpha]\nu \wedge \nu \models Q)
 \end{aligned}$$