Formal Methods for Security Gate Patterns

Mathematical Foundations for CVE-Surface Reduction

Author: Euman Engineer

Contact: opsec.ee@pm.me

February 2025

1 Gate Predicate Logic

Let **S** be the set of all observable program states. Let **Ω = { 0, 1, X, Z }** be the 4-state gate output domain:

```
0 ≡ PASS       precondition satisfied
1 ≡ FAIL       violation detected
X ≡ UNKNOWN    insufficient information to decide
Z ≡ INVALID    gate inputs themselves are corrupt
```

A gate **G** is a total function `G : S → Ω` satisfying the totality requirement `∀s ∈ S : G(s) ∈ Ω`. No gate may be partial. If the gate cannot evaluate, it returns X or Z — never ⊥.

GATE CLASS PREDICATES

Let `ptr ∈ Addr`, `rc ∈ ℕ`, `buf ∈ Byte*`, `τ ∈ Type`, `t ∈ Time`.

```
φ_UAF(ptr)    ≡  ptr ∈ Live  ∧  rc(ptr) > 0
φ_DF(ptr)     ≡  ptr ∈ Live  ∧  freed(ptr) = false
φ_NUL(ptr)    ≡  ptr ≠ 0x0   ∧  page(ptr) ∈ Mapped
φ_BOF(buf, i) ≡  0 ≤ i < len(buf)
φ_TYP(ptr, τ) ≡  typeof(ptr) ⊑ τ
φ_RAC(r, t)   ≡  ¬∃t′ ∈ (t_check, t_use) : mut(r, t′)
φ_REF(rc)     ≡  rc < RC_MAX  ∧  rc ≥ 0
```

Gate evaluation:

```
         ⎧ 0   if φ(s) = true
         ⎪ 1   if φ(s) = false
G(s)  =  ⎨ X   if φ(s) is undecidable at runtime
         ⎪ Z   if inputs to φ are themselves corrupt
         ⎩
```

UNIVERSAL INVARIANT

For any function **f** protected by gate set $\Gamma = \{G_1, \ldots, G_n\}$:

```
pre(f) ≡ ⋀ (Gᵢ(s) = 0)    for i = 1 to n
```

Execution of **f** proceeds if and only if all gates pass.

COMPOSITION ALGEBRA

Gate conjunction, where $\triangleright$ denotes sequential check:

```
(G₁ ▷ G₂)(s) = G₁(s)    if G₁(s) ≠ 0
              = G₂(s)    if G₁(s) = 0

Short-circuit:
  G₁(s) = 1  ⟹  (G₁ ▷ G₂)(s) = 1      fail-fast
  G₁(s) = Z  ⟹  (G₁ ▷ G₂)(s) = Z      corruption propagates
```

Ordering on **Ω** (failure lattice): $Z > 1 > X > 0$. For a gate vector $\Gamma = \langle G_1, \ldots, G_n \rangle$:

```
Γ(s) = max(G₁(s), G₂(s), …, Gₙ(s))
```

The worst violation dominates.

2 Object Lifetime State Machine

Use-after-free and double-free are illegal transitions in the object lifecycle. Gates enforce transition guards.

DEFINITION

```
M = (Q, Σ, δ, q₀, F)

Q  = { A, R, D, F, ⊥ }
  A = Allocated       memory obtained, not yet referenced
  R = Referenced      at least one live reference exists
  D = Dereferenced    all references released, rc = 0
  F = Freed           memory returned to allocator
  ⊥ = Error           illegal state — vulnerability triggered

q₀ = A    initial state
F  = { F }  accepting states
⊥ absorbing: no transitions out

Σ  = { alloc, ref, deref, free, access }
```

TRANSITION FUNCTION δ : Q × Σ → Q

```
δ(A, ref)    = R
δ(A, free)   = F
δ(A, deref)  = ⊥
δ(A, access) = ⊥

δ(R, ref)    = R            rc ← rc + 1
δ(R, deref)  = R            if rc − 1 > 0
δ(R, deref)  = D            if rc − 1 = 0
δ(R, access) = R
δ(R, free)   = ⊥            UAF-001 / CWE-416

δ(D, free)   = F
δ(D, ref)    = ⊥
δ(D, deref)  = ⊥
δ(D, access) = ⊥            UAF-001 / CWE-416

δ(F, alloc)  = A′           reallocation
δ(F, ref)    = ⊥            UAF-001 / CWE-416
δ(F, access) = ⊥            UAF-001 / CWE-416
δ(F, free)   = ⊥            DF-001  / CWE-415

δ(⊥, σ)     = ⊥            ∀σ ∈ Σ
```

GATE AS TRANSITION GUARD

For any transition `δ(q, σ)` where `δ(q, σ)` could reach ⊥:

```
δ_guarded(q, σ) = δ(q, σ)     if G_σ(s) = 0
                = q             if G_σ(s) ≠ 0     block and log
```

SAFETY THEOREM

If every transition into ⊥ is guarded by a gate G where G(s) is evaluated before δ, then: ∀ reachable states q : q ≠ ⊥

Proof. By induction on transition sequences. Base case: $q_0 = A \neq \perp$. Inductive step: assume $q_n \neq \perp$. If $\delta(q_n, \sigma) = \perp$, then G_σ fires and the transition is blocked, so $q_{n+1} = q_n \neq \perp$. ∎

REFCOUNT INVARIANT

For any object **o** with reference count `rc(o)`:

```
INV_REF(o)  ≡  0 ≤ rc(o) ≤ RC_MAX

where RC_MAX = 2³¹ − 1

REF-001 gate enforces:
  pre(ref)   :  rc(o) < RC_MAX          overflow prevention
  pre(deref) :  rc(o) > 0               underflow prevention
  pre(free)  :  rc(o) = 0  ∧  state(o) = D
```

BIJECTIVE CONTRACT

For a function `f : X → Y` protected by gate **G**:

```
{ G(s) = 0 }  f(x)  { G(s′) = 0 }
```

If the gate passes before **f** and **f** preserves the invariant, then the gate passes after **f**. This is the gate stability property — correct code cannot cause a gate to spontaneously transition from PASS to FAIL.

3 TOCTOU / RACE-001 Temporal Logic

Race conditions require reasoning over time. We formalize using linear temporal logic over discrete execution steps.

TEMPORAL MODEL

Let $T = \langle t_0, t_1, t_2, \dots \rangle$ be a discrete timeline of execution steps across all threads. For resource **r**:

```
val(r, t) = value of r at time t
mut(r, t) = true iff r is mutated at time t
```

TOCTOU WINDOW

A check-use pair `(c, u)` on resource **r** defines:

```
W(c, u) = { t ∈ T : t_c < t < t_u }

TOCTOU safety predicate:
  SAFE(r, c, u)  ≡  ∀t ∈ W(c, u) : ¬mut(r, t)

RACE-001 gate:
  φ_RAC(r, c, u) = SAFE(r, c, u)
```

LTL SPECIFICATION

Let `p` = resource checked, `q` = resource used, `m` = resource mutated.

Safety property:

```
□(p → (¬m U q))

Globally, if r is checked, then r is not mutated until r is used.
```

Violation witness:

```
◇(p ∧ ◇(m ∧ ¬q ∧ ◇q))

Eventually r is checked, then eventually mutated before use,
and then used with stale data.
```

WINDOW CONTRACTION THEOREM

Let `n` be the number of interleaving points in window `W(c, u)` and let `f_mut(r)` be the per-step probability of mutation on resource **r** by any thread θ_j $(j \neq i)$.

WINDOW CONTRACTION THEOREM

The probability of a TOCTOU violation on resource r over window W(c, u) is bounded by: $P(race) = 1 - (1 - f\_mut(r))^n$ where $n = |W(c, u)| = t\_u - t\_c$.

```
P(race) = 1 − (1 − f_mut(r))ⁿ

where n = |W(c, u)| = t_u − t_c     interleaving points

First-order approximation (f_mut(r) □ 1):
  P(race) ≈ n × f_mut(r)
```

Proof. At each interleaving point $t \in W(c, u)$, the probability that no mutation occurs is $(1 - f\_mut(r))$. The n points are independent scheduling decisions. Therefore $P(safe) = (1 - f\_mut(r))^n$ and $P(race) = 1 - P(safe) = 1 - (1 - f\_mut(r))^n$. By Bernoulli's inequality, for f_mut(r) □ 1: $(1 - f\_mut(r))^n \approx 1 - n \times f\_mut(r)$, yielding the linear bound. □

Corollary. P(race) is monotonically increasing in both n and f_mut(r). Reducing either reduces vulnerability exposure.

Mitigation strategies follow directly:

```
Strategy 1:  Minimize n              contract W
             n → 1  □  P(race) → f_mut(r)

Strategy 2:  Eliminate f_mut         hold mutex on r during W
             f_mut → 0  □  P(race) → 0

Strategy 3:  REAR verification       detect post-hoc
             does not reduce P(race), but guarantees detection:
             REAR asserts val(r, t_c) = val(r, t_u)
             violation □ RACE-001 fires retroactively
```

Strategies 1 and 2 reduce P(race) directly. Strategy 3 accepts the race window but guarantees no undetected exploitation — the REAR sidecar provides the completeness bound.

INTERLEAVING SEMANTICS

Let $\theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ be the set of active threads. RACE-001 is violated iff:

```
∃σ ∈ Interleavings(θ) :
    ∃(c, u) ∈ σ|θᵢ,  ∃m ∈ σ|θⱼ  (i ≠ j) :
        t_c < t_m < t_u  ∧  target(m) = target(c)
```

Gate coverage: a gate set **Γ** is RACE-COMPLETE for resource set **R** iff:

```
∀r ∈ R, ∀(c, u) on r :  ∃G ∈ Γ : G validates SAFE(r, c, u)
```

---

4 Overhead Cost Model

Security levels form a lattice. Gate activation is monotonic over this lattice. The compile-out property guarantees zero cost at the bottom.

SECURITY LEVEL LATTICE

```
L = { NONE, BASIC, STANDARD, PARANOID }

NONE  ⊑  BASIC  ⊑  STANDARD  ⊑  PARANOID

L forms a total order (lattice with ⊓ = min, ⊔ = max).
```

GATE ACTIVATION FUNCTION

Let □□ = {UAF, DF, NULL, BOF, TYPE, RACE, REF} be all gate classes. Define activation α : L → □□(□□):

```
α(NONE)     = ∅
α(BASIC)    = { REF, TYPE }
α(STANDARD) = { REF, TYPE, BOF, NULL }
α(PARANOID) = □□         all gates plus canaries

Monotonicity:  l₁ ⊑ l₂  □  α(l₁) ⊆ α(l₂)
```

COST FUNCTION

```
Cost(f, l) = fᵢ  ×  Σ{ cⱼ : Gⱼ ∈ Γ_f ∩ α(l) }

Overhead(l) = Σ_f Cost(f, l)  /  Baseline

Measured bounds:
  Overhead(NONE)      =  0%       proven below
  Overhead(BASIC)     ≈  5%       REF + TYPE only
  Overhead(STANDARD)  ≈ 10%       plus bounds checks
  Overhead(PARANOID)  ≈ 20%       all gates + memory canaries
```

COMPILE-OUT THEOREM

Claim: Overhead(NONE) = 0 exactly.

Proof. *α(NONE) = □ by definition. ⟹ ∀f : Γ_f ∩ α(NONE) = □ (intersection with □). ⟹ ∀f : Cost(f, NONE) = 0 (empty sum). ⟹ Overhead(NONE) = 0/B = 0. □*

Implementation: `GATE_CHECK(id, …)` expands to `((void)0)` when `SECURITY_LEVEL = NONE`. Any C++ optimizer with dead code elimination removes this entirely.

COST ORDERING

```
l₁ ⊑ l₂   □  Overhead(l₁) ≤ Overhead(l₂)
```

Cost is monotonically non-decreasing with security level. You pay strictly for what you enable.

---

5 Sidecar Composition

The three-thread sidecar model as a formal pipeline with verification completeness.

PIPELINE MODEL

```
FRONT : S → S × Hint
LEAD  : S × Hint → S′
REAR  : S × S′ → { ✓, ✗ }

Pipeline composition:
  (h, s₁) = FRONT(s₀)        predict and prefetch
  s₂      = LEAD(s₁, h)       authoritative execution
  result  = REAR(s₁, s₂)      postcondition verification

FRONT is advisory: ∀h : LEAD(s, h) = LEAD(s, ⊥)
Hints improve performance, not correctness.
```

REAR VERIFICATION SEMANTICS

REAR checks a set of postconditions $\Pi = \{\pi_1, \dots, \pi_k\}$:

```
REAR(s, s′) = □ πᵢ(s, s′)    for i = 1 to k

π_UAF(s, s′)  ≡  ∀ptr freed in s→s′ : rc(ptr, s) = 0
π_BOF(s, s′)  ≡  ∀write(buf, i) in s→s′ : i < len(buf, s)
π_RAC(s, s′)  ≡  ∀(c, u) in s→s′ : val(r, t_c) = val(r, t_u)
π_TYPE(s, s′) ≡  ∀cast(ptr, τ) in s→s′ : typeof(ptr, s) ⊑ τ
```

VERIFICATION COMPLETENESS

REAR is complete for gate set □□ iff:

```
∀G ∈ □□, ∀(s, s′) :
    G would have returned 1 on some sub-state of s→s′
    □  REAR(s, s′) = ✗
```

DEFENSE IN DEPTH THEOREM

Let Γ_pre be gates checked before LEAD. Let Π_post be REAR postconditions. If Π_post is complete for □□, then: ∀ vulnerability v ∈ □□ : v is caught by Γ_pre (before damage) ∨ v is caught by Π_post (after execution, before propagation). No vulnerability in the gate taxonomy escapes both layers.

VIEWER-SPECIFIC MAPPING

```
FRONT  →  Interest List / LOD       h = prefetch_hints(s)
LEAD   →  Main Loop (render, sim)   s′ = execute(s, h)
REAR   →  (does not exist yet)      result = verify(s, s′)
```

Composition guarantee:

```
REAR(LEAD(s, FRONT(s)))  ▯  ▯ π ∈ Π
```

When REAR returns ▯: gate fires retroactively, state is rolled back or quarantined, and the violation is logged with full `(s, s′, π)` context.

Notation Reference

∀	for all	∃	there exists
∧	logical AND	∨	logical OR
¬	logical NOT	▯	implies
▯	if and only if	∈	element of
⊆	subset of	⊑	subtype of
∅	empty set	▯▯(X)	power set of X
▯	conjunction	▯	satisfies / models
□	always (LTL)	◇	eventually (LTL)
U	until (LTL)	⊥	bottom / undefined
∘	fn composition	▯	parallel composition
▷	sequential gate	∎	end of proof
⊓	lattice meet	⊔	lattice join
ℕ	natural numbers	Ω	gate output domain
α	gate activation	δ	transition function
σ	execution trace	Θ	thread set
Γ	gate set	Π	postcondition set
φ	gate predicate	π	postcondition pred