

What a Mutation Regime Can Know About a Function

Anonymous Author(s)
Affiliation

Abstract—Mutation testing measures a test suite by the fraction of a fixed family of mutant programs it distinguishes, and a score of one is read as evidence of adequacy — but the field has never characterized what such a score actually establishes about the program under test. We give that characterization for output mutation (perturb the returned value; extreme/Descartes-style mutation is the constant case), and cast it as a theory of knowability: which mutation regime is provably sufficient to determine a function’s output behavior, exactly how much of that behavior a passing score determines, and precisely where mechanical determination ends and a human intent test begins. For adequacy an output operator is exactly its footprint — the set of values it changes — and we prove (Theorem 3.2, machine-checked) that a finite regime Π is sufficient to certify a target class Γ , uniformly over all programs and suites, iff every target footprint is the union of the Π -footprints it contains (an infinite regime obeys a finite-avoidance form of the same law). The consequences form a complete profile of mechanical knowability. The minimal sufficient regime on n output values is the n singleton value guards, and Descartes constants are sufficient iff $n = 2$ (extreme mutation is provably optimal on Booleans and provably deficient beyond). A full score knows exactly output coverage — that the suite exercised every value the program can return — and nothing else: the ceiling of the technique (Theorem 4.5). Mechanical sufficiency is achievable only on finite return types and decidable exactly when footprint containment is — polynomial for value tables, decidable for semilinear footprints, undecidable in general (Theorem 5.2). Under a weak oracle the boundary relativizes and yields an exact mixed-oracle characterization of the pseudo-tested method (§9), whose oracle-dependence an empirical study of five Python libraries illustrates: 43% of covered methods are pseudo-tested under a value oracle against 6% under a crash-inclusive one. Finally, what the regime cannot know from the program alone — which of the distinctions it certifies bear on the program’s intent — is the residual a human supplies with direct, authored intent tests (§10). The formal core is machine-checked in Lean 4 / Mathlib, end to end from programs to the kernel.

Index Terms—Mutation testing, test adequacy, operator completeness, extreme mutation, interactive theorem proving.

I. Introduction

A. The question

A mutation-adequacy tool seeds a program with a small syntactic fault and reports whether the suite kills it — distinguishes the mutant from the original on some input. A score of one is the field’s adequacy certificate. Its force rests entirely on the operator family: the score certifies the suite against exactly the faults the operators can express.

Two questions have never had clean answers, and they are really one question — what does a passing score let you know about the function?

First, what does a score of one determine? Not correctness; only that no seeded fault survived. The bridge from “no seeded fault survives” to “no real fault survives” is the coupling-effect and competent-programmer hypotheses (DeMillo, Lipton & Sayward 1978), stated as motivating assumptions and never proven. Second, which operator family is enough? The strongest prior results are relative by construction: sufficient operator sets ([13]) are found experimentally against a fixed pool; minimal mutant sets and dominator mutants ([1]) are minimal relative to a fixed test set and pool; true semantic mutant subsumption — the order a completeness theorem would rank against — is undecidable ([9]); and the foundational formal treatment ([2]) establishes undecidability of related decision problems rather than a characterization.

The relativity is forced by the object. A program-text mutation schema carries no operator-only invariant that fixes its detection uniformly across programs — the mutant it produces depends on the text of the program, not on the function it computes — so “does schema X subsume schema Y ” has no program-independent answer (§7). This is why a completeness theory has been unavailable, and it points to the sub-class where one is available.

B. The frame: mechanical knowledge and the intent residual

We restrict to output mutation: operators that perturb the returned value of a function, independent of how it is computed. For this class we ask not which faults a regime can express but what killing its mutants determines, uniformly over all programs and suites — and we find that the answer partitions cleanly into two layers.

- The mechanically knowable. How a function’s outputs are distributed — which distinct values it

produces, and whether a suite has observed them — is determined by a suitable mutation regime, decidable, up to an exact ceiling. This is the full profile of what a tool can establish about output behavior without being told what the program is for. It is automatable.

- The intent residual. Which of the distinctions the regime certifies actually matter — bear on what the program is supposed to do — is not determined by the program text. It is supplied by authored intent,

through constructed tests, and it is exactly what a weak oracle leaves unpinned (§9–§10).

This paper characterizes the first layer completely and locates the boundary to the second precisely.

C. Contributions

- 1) The footprint reduction (§2): for adequacy an output operator is its footprint, so a regime’s power is a family of subsets of the return type.
- 2) The sufficiency characterization (§3, Theorem 3.2, machine-checked): a regime Π certifies a target class Γ uniformly over all programs and suites iff every target footprint is the union of the Π -footprints it contains.
- 3) The minimal sufficient regime and the ceiling (§4): the n value guards are the minimal absolutely-sufficient regime; constants are sufficient iff $n = 2$; a full score knows exactly output coverage.
- 4) Where knowability is achievable and decidable (§5): mechanical sufficiency exists only on finite return types, and deciding it is P / decidable / undecidable through footprint containment.
- 5) What the regime does not determine from structure (§6–§7): coupling is not entailed for output mutation; detection factors through the footprint, which is why the theory is program-independent.
- 6) The cost of the knowledge (§8): the minimum certifying suite has size $|f(D)|$, the teaching dimension.
- 7) The oracle-relative boundary and the intent residual (§9–§10): the whole development relativizes to a weak oracle and characterizes the pseudo-tested method — the mechanically-observed distinction a human intent test must pin — with an empirical measurement.

The formal core (§2–§8) is machine-checked in Lean 4 / Mathlib, from real programs $f : D \rightarrow R$ and finite suites down to the kernel (§11), and is provided in supplemental materials.

II. The model

Definition 2.1 (program, suite, output operator). A program computes a total function — its denotation — $f : D \rightarrow R$, from inputs D to a return type R with $|R| \geq 2$. A suite is a finite $T \subseteq D$. An output operator is a total map $p : R \rightarrow R$; it produces the mutant $p \circ f$ (run the program, then perturb the result). A regime Π is a set of output operators.

Definition 2.2 (footprint). The footprint of p is $\text{Mov}(p) = \{r \in R : p(r) \neq r\}$, the values it changes. Write $I = f(D)$ for the program’s reachable outputs and $O = f(T) \subseteq I$ for the suite’s observed outputs.

Definition 2.3 (equivalence, killing, score). $p \circ f$ is equivalent iff $p(f(x)) = f(x)$ for all $x \in D$; T kills it iff $p(f(x)) \neq f(x)$ for some $x \in T$; the score $\text{score}_\Pi(f, T) = 1$ iff T kills every non-equivalent $p \circ f$, $p \in \Pi$.

Proposition 2.4 (an operator is its footprint). For any f , T , p :

$$\begin{aligned} p \circ f \text{ equivalent} &\iff \text{Mov}(p) \cap I = \emptyset, \\ T \text{ kills } p \circ f &\iff \text{Mov}(p) \cap O \neq \emptyset. \end{aligned}$$

So whether p is a non-equivalent survivor depends only on $\text{Mov}(p)$; equal footprints are interchangeable, and $\text{score}_\Pi(f, T)$ depends only on $\{\text{Mov}(p) : p \in \Pi\}$.

Proof. $p \circ f$ is equivalent iff p fixes every element of $I = f(D)$, i.e. $\text{Mov}(p) \cap I = \emptyset$. T kills $p \circ f$ iff some $x \in T$ has $f(x) \in \text{Mov}(p)$, i.e. $\text{Mov}(p) \cap f(T) = \text{Mov}(p) \cap O \neq \emptyset$. Both read only $\text{Mov}(p)$ against the fixed sets I, O . ■

Remark 2.5 (every subset is a footprint). For $|R| \geq 2$ every $S \subseteq R$ is some operator’s footprint: fix $a \neq b$, let $q(a) = b$ and $q(x) = a$ otherwise (fixed-point-free), and set $p(x) = q(x)$ for $x \in S$, $p(x) = x$ otherwise. So footprints range over the entire powerset, and the adequacy content of a regime Π is exactly the family $\text{Foot}(\Pi) = \{\text{Mov}(p) : p \in \Pi\}$. Where an operator sends the values it moves is invisible to adequacy; this underlies everything below.

The knowability question, stated. By Proposition 2.4, running a regime Π on a program tells you exactly which footprints in $\text{Foot}(\Pi)$ meet the reachable set I (which mutants are non-equivalent) and which meet the observed set O (which are killed). What a passing score lets you know about f is therefore a question about set systems — the subject of §3.

Remark 2.6 (scope, and where the boundary to intent first appears). §2–§8 assume an exact output oracle: a mutant is killed exactly when a test yields a different return value (Definition 2.3), i.e. the suite observes the full output. This is the load-bearing assumption, and it is exactly where the intent residual enters: under a weaker oracle a covered mutant can survive because no assertion inspects the changed value — the pseudo-tested method — and §9 relativizes the whole theory to make this precise. Two simplifying assumptions: denotations are pure total maps (void methods, exceptions, effects are out of scope), and an operator perturbs every returned value identically (a statement-level mutation of one return site is not of this form).

III. The sufficient regime

Definition 3.1 (sufficiency). A regime Π is sufficient for a target class Γ iff for every program $f : D \rightarrow R$ and every finite suite T ,

$$\text{score}_\Pi(f, T) = 1 \implies \text{score}_\Gamma(f, T) = 1.$$

Π is absolutely sufficient iff it is sufficient for $\Gamma = \mathcal{T}(R)$, all output operators. The quantifier over all (f, T) removes both the program-text and the suite relativity of prior completeness notions: sufficiency asks what a kill certifies as such, not on some corpus. Here D varies — the quantifier ranges over all input domains and all $f : D \rightarrow R$;

by Proposition 2.4 what matters is only the pair $(I, O) = (f(D), f(T))$ with $O \subseteq I$, and §11’s realizability shows every such pair is met by an actual program, so quantifying over programs and over (I, O) pairs coincide. The separating program built in Theorem 3.2’s proof uses a finite domain of size $|D| = |A| + 1 \leq |\Pi| + 1$ (one escape point per operator that moves b , plus b itself).

Theorem 3.2 (footprint characterization). Let Π be finite. Then Π is sufficient for Γ iff

$$\forall g \in \Gamma, \forall b \in \text{Mov}(g), \exists p \in \Pi : b \in \text{Mov}(p) \subseteq \text{Mov}(g),$$

equivalently: every target footprint is the union of the Π -footprints it contains.

Proof. ($\Leftarrow$) Assume the condition; fix (f, T) with $\text{score}_{\Pi}(f, T) = 1$ and non-equivalent $g \in \Gamma$. By Proposition 2.4 pick $b \in \text{Mov}(g) \cap I$; take $p \in \Pi$ with $b \in \text{Mov}(p) \subseteq \text{Mov}(g)$. Then $b \in \text{Mov}(p) \cap I$, so $p \circ f$ is non-equivalent, hence killed: $\text{Mov}(p) \cap O \neq \emptyset$. Since $\text{Mov}(p) \subseteq \text{Mov}(g)$, also $\text{Mov}(g) \cap O \neq \emptyset$, so T kills $g \circ f$.

($\Rightarrow$) By contraposition. Suppose the condition fails at $g \in \Gamma$, $b \in \text{Mov}(g)$: every $p \in \Pi$ with $b \in \text{Mov}(p)$ has an escape point $a_p \in \text{Mov}(p) \setminus \text{Mov}(g)$. Let $A = \{a_p\}$; finite (as Π is), with $b \notin A$. Take $I^* = \{b\} \cup A$, $D = I^*$, $f = \text{id}$ (so $I = I^*$), and $T = A$ (so $O = A$). Every non-equivalent Π -mutant dies: if $b \in \text{Mov}(p)$ then $a_p \in \text{Mov}(p) \cap O$; if $b \notin \text{Mov}(p)$ then $\text{Mov}(p) \cap I = \text{Mov}(p) \cap A \neq \emptyset$. But $g \circ f$ survives: $b \in \text{Mov}(g) \cap I$ so it is non-equivalent, while $\text{Mov}(g) \cap O = \text{Mov}(g) \cap A = \emptyset$. So $\text{score}_{\Pi} = 1 \not\equiv \text{score}_{\Gamma} = 1$. ■

Theorem 3.2b (general Π). Finiteness of Π is used in Theorem 3.2 only to keep the escape set A finite; without it the containment $\text{Mov}(p) \subseteq \text{Mov}(g)$ weakens to a finite-avoidance condition. For arbitrary Π , Π is sufficient for Γ iff

$$\forall g \in \Gamma, \forall b \in \text{Mov}(g), \forall F \subseteq R \setminus \text{Mov}(g) \text{ finite}, \\ \exists p \in \Pi : b \in \text{Mov}(p) \wedge \text{Mov}(p) \cap F = \emptyset.$$

For finite Π this reduces to Theorem 3.2 (take F to be the escape set A). The two forms differ exactly on infinite families over an infinite R : on $R = \mathbb{N}$ the regime $\Pi = \{\{0, k\} : k \geq 1\}$ is sufficient for the guard target $\text{Mov}(g) = \{0\}$ — for any finite $F \subseteq \mathbb{N} \setminus \{0\}$ some $k \notin F$ gives $\{0, k\} \cap F = \emptyset$ — yet Π contains no footprint $\subseteq \{0\}$, so Theorem 3.2’s condition fails while 3.2b’s holds. Theorem 3.2 is therefore the finite- Π instance, not the general law; the finiteness hypothesis is load-bearing and is stated wherever Theorem 3.2 is invoked below. Both forms are machine-checked (footprint_characterization, footprint_characterization_general; §11).

Remark 3.3 (composition is the wrong closure). Theorem 3.2 dissolves the temptation to define sufficiency by generation under composition. Killing a regime’s operators need not kill their composites: on $R = \{0, 1, 2\}$ with sole reachable/observed value 0, and $a(0) = 1$, $b(0) = 2$, $b(1) = 0$,

both $a \circ f$ and $b \circ f$ die at 0, yet $(b \circ a)(0) = b(1) = 0$ — so $(b \circ a) \circ f$ is equivalent and unkillable. Composition acts on exactly the part of an operator adequacy cannot see ($\text{Mov}(b \circ a)$ can be strictly smaller than both), so “ Π generates the transformation monoid” says nothing about what a score certifies. The governing order is footprint containment, not compositional generation.

IV. The minimal sufficient regime, and the ceiling

Corollary 4.1 (the value-guard basis; minimal size n). Let $|R| = n$. A regime Π is absolutely sufficient iff it contains, for every $r \in R$, an operator with footprint $\{r\}$ — a value guard γ_r , “if the result is r , return something else.” The minimal absolutely-sufficient regime is $\{\gamma_r : r \in R\}$, of size exactly n .

Proof. Absolute sufficiency is sufficiency for $\mathcal{T}(R)$, whose footprints are all of 2^R (Remark 2.5). Apply Theorem 3.2b to the singleton target $\text{Mov}(g) = \{b\}$ with $F = R \setminus \{b\}$ — finite, because R is: sufficiency forces some $p \in \Pi$ with $b \in \text{Mov}(p)$ and $\text{Mov}(p) \cap (R \setminus \{b\}) = \emptyset$, i.e. $\text{Mov}(p) = \{b\}$, a value guard for b . (Because R is finite the whole complement is an admissible F , so the singleton is forced for any Π — the infinite- Π escape of Theorem 3.2b is unavailable on a finite return type, which is why the “iff” needs no finiteness hypothesis on Π here even though Theorem 3.2 does.) Conversely a guard per value satisfies 3.2b’s condition for every target. Each of the n singleton targets forces a distinct guard, so no smaller regime suffices. ■

Corollary 4.2 (mechanical sufficiency requires a finite return type). A finite regime can be absolutely sufficient only when R is finite; for an infinite return type — int, String, objects — no finite output-mutation regime is absolutely sufficient.

Proof. By Theorem 3.2 with Π finite (the hypothesis here) — Corollary 4.1 is stated for finite R and cannot be invoked on an infinite return type: absolute sufficiency forces a distinct singleton footprint per value, since the singleton target $\{b\}$ demands some $p \in \Pi$ with $b \in \text{Mov}(p) \subseteq \{b\}$, i.e. $\text{Mov}(p) = \{b\}$. As $r \mapsto \{r\}$ is injective, a sufficient regime has $\geq |R|$ operators. ■

Corollary 4.2 draws the first knowability boundary: complete mechanical knowledge of output behavior is achievable only on finite return types (Booleans, enums, small tagged unions). On the return types of most real programs the absolute regime is unavailable, and the meaningful object is relative sufficiency for a chosen footprint class (§5) — the certificate a tool can actually offer.

Corollary 4.3 (extreme mutation is optimal on Booleans, deficient beyond). A constant operator return c has footprint $R \setminus \{c\}$. The constant regime is absolutely sufficient iff $n = 2$: then return true/return false have footprints

$\{\text{false}\}/\{\text{true}\}$ — exactly the two guards — while for $n \geq 3$ every constant footprint has size ≥ 2 , so no constant is a guard.

Example 4.4 (a distinction the constants cannot know). Let $R = \{a, b, c\}$, a program reaching all three, a suite observing only $\{a, b\}$. All three constants die, so the constant score is 1. But the fault “returns b where it should return c ” — footprint $\{c\}$ — is non-equivalent ($c \in I$) and unkillable ($\{c\} \cap O = \emptyset$). A perfect extreme-mutation score determines nothing about the c -outputs the suite never observed.

Theorem 4.5 (the ceiling — exactly what a full score knows). For $|R| \geq 2$ and any f, T :

$$\text{score}_{\mathcal{T}(R)}(f, T) = 1 \iff I \subseteq O \iff O = I,$$

i.e. an absolute score of one holds exactly when the suite exercises every reachable output. This is the whole of what output mutation determines, and no more.

Proof. ($\Leftarrow$) If $O = I$, every non-equivalent g has $\text{Mov}(g) \cap I \neq \emptyset$, hence $\text{Mov}(g) \cap O \neq \emptyset$, so all die. ($\Rightarrow$) If $I \not\subseteq O$, the guard γ_b for $b \in I \setminus O$ is non-equivalent and unkillable, so the score is below one. ■

Remark 4.6 (the ceiling, and the first face of the intent residual). Theorem 4.5 is the operational content of output mutation: a full score certifies output coverage — every value the program can return was observed — and the value guards are the cheapest regime making that exact. It knows nothing about faults separating two inputs mapped to the same output: if $f(x_1) = f(x_2)$ then $p(f(x_1)) = p(f(x_2))$ for every output operator, so no output mutant distinguishes them. Input-separating behavior is structurally outside the regime’s knowledge — the first, purely structural, piece of what the mechanical layer cannot reach.

V. Where knowability is decidable

Fix a finite Π and a finitely-presented target Γ . By Theorem 3.2, deciding sufficiency is deciding footprint containment, and its complexity is exactly that of the footprint representation.

Proposition 5.1 (the exact reach of a partial guard regime). The guards $\{\gamma_{r_1}, \dots, \gamma_{r_k}\}$ are sufficient for exactly $\Gamma_{\max} = \{g : \text{Mov}(g) \subseteq \{r_1, \dots, r_k\}\}$.

Proof. By Theorem 3.2 the guards are sufficient for g iff every $b \in \text{Mov}(g)$ has a guard γ_{r_i} with $b \in \{r_i\} \subseteq \text{Mov}(g)$, i.e. $b \in \{r_1, \dots, r_k\}$; this holds for all $b \in \text{Mov}(g)$ iff $\text{Mov}(g) \subseteq \{r_1, \dots, r_k\}$. ■

Proposition 5.1 is the usable form: a partial guard regime yields an explicit, checkable certificate — every guarded value the program can produce is exercised, hence any fault confined to the guarded values is caught — with the guarded set a named parameter.

Theorem 5.2 (decidability spectrum). Deciding whether finite Π is sufficient for finite Γ reduces to finitely many footprint-containment tests $\text{Mov}(p) \subseteq \text{Mov}(g)$ and membership tests. Hence: 1. Finite return type (value tables): footprints are explicit subsets; each test is a linear scan, so sufficiency is decidable in polynomial time. 2. Semilinear footprints ($R = \mathbb{Z}$, Presburger-definable Mov): containment is decidable, so sufficiency is decidable. 3. Total computable operators: each footprint is a decidable set, but deciding containment — hence sufficiency — is undecidable.

Proof. The reduction is Theorem 3.2. (1) tables give linear containment; (2) Presburger arithmetic is decidable; (3) reduce from non-halting between two non-trivial operators. For a machine M , the total computable $g_M(k) = k + 1$ if M halts within k steps and $g_M(k) = k$ otherwise has decidable footprint $\text{Mov}(g_M) = \{k : M \text{ halts within } k \text{ steps}\}$ — which is $\emptyset$ if M never halts and a final segment $[t, \infty)$ if M halts at step t (halting within k steps is monotone in k). Let h be the total computable operator with footprint $\text{Mov}(h) = \{k : k \text{ odd}\}$ — a non-degenerate footprint. Then $\text{Mov}(g_M) \subseteq \text{Mov}(h)$ iff $\text{Mov}(g_M) = \emptyset$ (a final segment $[t, \infty)$ contains even numbers, so is never a subset of the odds), i.e. iff M never halts. That containment is the condition for $\{h\}$ to be sufficient for $\{g_M\}$ (the Π -footprint $\text{Mov}(h)$ sits inside the target $\text{Mov}(g_M)$, per Theorem 3.2); the reverse question, sufficiency of $\{g_M\}$ for $\{h\}$, would need the odds $\subseteq \text{Mov}(g_M)$ and is decidable false. So sufficiency of $\{h\}$ for $\{g_M\}$ — containment between two non-trivial computable footprints — is undecidable. ■

Remark 5.3 (the spectrum runs through set containment). The P / decidable / undecidable gradient is that of subset containment for increasingly expressive set representations. No word problems or Cayley embeddings appear, because adequacy never composes operators (Remark 3.3): it is the containment order, not a generation order, that governs whether knowability is decidable. So the second knowability boundary — is the sufficiency question itself answerable — is exactly the decidability of footprint containment.

VI. What structure alone cannot determine

Proposition 6.1 (coupling is not entailed). There exist a regime Π , program f , and suite T under which every non-equivalent first-order Π -mutant is killed while a non-equivalent higher-order mutant survives.

Proof. Let $R = \{0, 1, 2\}$, $\Pi = \{a\}$ with $a : 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto 1$ (so $\text{Mov}(a) = R$), and f with $I = \{0, 2\}$, $O = \{0\}$. The first-order mutant $a \circ f$ is killed at 0. But $a \circ a : 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0$ has $\text{Mov}(a \circ a) = \{2\}$: non-equivalent ($2 \in I$), unkillable ($2 \notin O$). So the score is 1 against $\{a\}$ and below 1 against its composition closure. ■

The coupling-effect hypothesis — that killing first-order mutants tends to kill higher-order ones — is therefore not a theorem for output mutation. Proposition 6.1 is Remark 3.3 read against the coupling hypothesis rather than against generation: the same fact — a composite’s footprint can be strictly smaller than its factors’ — that makes compositional generation the wrong closure makes first-order kills not force higher-order ones. By Theorem 3.2 higher-order-ness carries no special weight: a higher-order mutant is simply an operator with a footprint, and the closure may contain footprints the regime does not cover. We claim only that coupling is not entailed — not that it fails to occur, and not that anyone advanced it as a theorem for output mutation; the hypothesis is a statistical claim about typical faults, which a single counterexample does not refute. The content is the mechanism — footprint shrinkage under composition — not a defeat of the hypothesis.

VII. Why the theory is program-independent

Proposition 7.1 (program-independent subsumption). For output operators p, g : every test that kills $p \circ f$ also kills $g \circ f$, for every program f , iff $\text{Mov}(p) \subseteq \text{Mov}(g)$.

Proof. ($\Leftarrow$) A test x kills $p \circ f$ iff $f(x) \in \text{Mov}(p) \subseteq \text{Mov}(g)$, so it kills $g \circ f$. ($\Rightarrow$) If $b \in \text{Mov}(p) \setminus \text{Mov}(g)$, a program reaching b with a suite observing b kills $p \circ f$ but not $g \circ f$. ■

So Theorem 3.2 reads, in the language of mutant subsumption: Π is sufficient for Γ iff every target mutant is subsumed by some Π -mutant, uniformly over all programs — the program-independent form of dominator theory.

Proposition 7.2 (the factoring that buys program-independence). For an output operator the detection set factors through the footprint: $\text{Det}(p \circ f) = \{x : p(f(x)) \neq f(x)\} = f^{-1}(\text{Mov}(p))$. So p enters detection only through the program-independent object $\text{Mov}(p)$. The contrast with program-text mutation is not that its detection is syntactic — any mutant’s detection set is semantic — but that a program-text schema carries no operator-only invariant: its mutant depends on the text of f , so its detection is a function of the (schema, source) pair, not the schema alone, and varies across programs computing the same f . This is why the field’s results are relative by construction, and why the knowability theory of §3 is available for output mutation and not for program-text mutation.

VIII. The cost of the knowledge

Theorem 8.1 (minimum certifying suite). Fix an absolutely-sufficient regime and a program f with reachable set $I = f(D)$. A suite T achieves $\text{score}(f, T) = 1$ iff $O = I$. If I is finite, the minimum certifying suite has size $\sigma(f) = |I| = |f(D)|$ (one test per reachable value). If I is infinite, no finite suite is certifying: finite T has finite $O = f(T)$, so $O \neq I$.

Proof. By Theorem 4.5 the score is 1 iff $O = I$. If I is finite, a suite with $O = I$ has, for each $r \in I$, some x with $f(x) = r$, so $|T| \geq |I|$; one x per value gives $|T| = |I|$. If I is infinite, $O = f(T)$ is finite for finite T . ■

Remark 8.2 (the sample complexity of knowledge). $\sigma(f)$ is the teaching dimension of the program against the regime (Goldman & Kearns 1995): the minimum labeled examples — tests with their observed outputs — that distinguish f from every non-equivalent mutant the regime expresses. For output mutation it is exactly the number of reachable outputs. Theorem 3.2 says which faults a regime can certify against; Theorem 8.1 says how many tests the certification costs. The mechanically-knowable profile of a function’s output behavior is therefore not only characterized but priced.

IX. The oracle-relative boundary and the pseudo-tested method

The exact oracle (Remark 2.6) is the model’s one load-bearing simplification, and the one that most bounds practical relevance — because the phenomenon that motivates extreme mutation, the pseudo-tested method, is a weak-oracle phenomenon by construction. This section relativizes the theory to an arbitrary oracle and shows the intent residual is exactly what a weak oracle leaves unpinned.

Definition 9.1 (oracle). An oracle is a map $\omega : R \rightarrow \Omega$ recording what the suite’s assertions observe about a returned value. The exact oracle is $\omega = \text{id}$; a suite checking only a field, a hash, a type tag, or nothing is a non-injective ω .

Definition 9.2 (oracle-relative detection). T ω -kills $p \circ f$ iff $\omega(p(f(x))) \neq \omega(f(x))$ for some $x \in T$; $p \circ f$ is ω -equivalent iff equality holds for all x . The oracle-relative detection set is $\text{Det}_\omega(p) = \{r : \omega(p(r)) \neq \omega(r)\}$.

Proposition 9.3 (the relativized reduction). $p \circ f$ is ω -equivalent iff $\text{Det}_\omega(p) \cap I = \emptyset$; T ω -kills it iff $\text{Det}_\omega(p) \cap O \neq \emptyset$. Moreover $\text{Det}_\omega(p) \subseteq \text{Mov}(p)$, with equality for injective ω . Theorems 3.2, 3.2b, 4.5, 5.2, 8.1 and Propositions 5.1, 7.1 hold with Mov replaced by Det_ω throughout — each reads a footprint only as a set against I and O , which Det_ω also is. The one result that does not relativize verbatim is Corollary 4.3 (Remark 9.3b).

Proof. Identical to Proposition 2.4 reading $\omega \circ p$ against ω ; containment is immediate. ■

Remark 9.3b (the footprint abstraction partially gives way). Under a non-injective ω , $\text{Det}_\omega(p)$ reads $\omega \circ p$, so where p sends the values it moves — invisible to adequacy under the exact oracle (Remark 2.5) — becomes visible again: two operators with equal Mov can have different Det_ω . Corollary 4.3 is where this bites. A constant has $\text{Det}_\omega(\text{return } c) = R \setminus \omega^{-1}(\omega(c))$, so the constant regime is absolutely ω -sufficient iff $|R| = 2$ and ω separates

the two values ($\omega(\text{true}) \neq \omega(\text{false})$) — the extra clause the exact-oracle statement (Corollary 4.3) does not carry, because a constant landing in ω 's own class of the value it replaces is ω -equivalent. So "extreme mutation is optimal on Booleans" survives relativization only for an oracle that actually inspects the boolean.

The interesting content is where Mov and Det_ω differ. The ceiling does not lower under a weak oracle: for any non-constant ω and any value b there is an operator with $\text{Det}_\omega(p) = \{b\}$, so the absolute ω -score is 1 iff $I \subseteq O$ — output coverage remains exactly what a full score knows, provided the suite asserts something. What weakening the oracle changes is which regimes reach the ceiling, and it is what makes a covered mutant survive.

Proposition 9.4 (pseudo-testedness, characterized). Let f be covered by T (so $O \neq \emptyset$). The constant mutant return c is pseudo-tested — it changes a value the program genuinely returns, yet no test in T detects the change under oracle ω — iff

$\text{Mov}(\text{'return } c') \cap I \neq \emptyset$ and $\text{Det}_\omega(\text{'return } c') \cap O = \emptyset$, equivalently $I \neq \{c\}$ (some reachable output is not c , so the constant is non-equivalent under the exact oracle) and $O \subseteq \omega^{-1}(\omega(c))$ (every observed output is oracle-indistinguishable from c , so the weak oracle sees no change).

The two conditions read non-equivalence and survival under different oracles, and that is the crux. Non-equivalence must be exact-oracle: the paradigm pseudo-tested method is one whose output no assertion inspects at all, so every reachable output lies in c 's ω -class ($I \subseteq \omega^{-1}(\omega(c))$) and the mutant is ω -equivalent — yet its return value genuinely differs from the original on some input ($\text{Mov} \cap I \neq \emptyset$), which is exactly the fault the weak oracle fails to catch. Requiring ω -non-equivalence (the natural but wrong reading, $\text{Det}_\omega \cap I \neq \emptyset$) would exclude this case — the very phenomenon the notion names. Under the exact oracle $\omega = \text{id}$ we have $\text{Det}_\omega = \text{Mov}$ and the two conditions collapse to $O = \{c\}$ against $I \neq \{c\}$: the suite observes only c while the program returns more.

Proof. ω -survival is $\text{Det}_\omega(\text{'return } c') \cap O = \emptyset$, i.e. $O \subseteq \omega^{-1}(\omega(c))$. Exact-oracle non-equivalence is $\text{Mov}(\text{'return } c') \cap I \neq \emptyset$; since $\text{Mov}(\text{'return } c') = R \setminus \{c\}$ this is $I \neq \{c\}$. Under $\omega = \text{id}$, $\text{Det}_\omega = \text{Mov}$ and $\omega^{-1}(\omega(c)) = \{c\}$. ■

This is the model's account of the empirical phenomenon, and the precise location of the intent residual. A method is pseudo-tested exactly when the suite's assertions collapse everything it observes into one oracle class, so replacing the body with a constant in that class changes nothing the suite can see. The regime knows, mechanically, that the returned value changed (the effect, $\text{Mov} \neq \emptyset$); what it cannot know is whether that change

matters — and the gap $\text{Mov}(p) \setminus \text{Det}_\omega(p)$ is exactly the oracle weakness a human closes by writing an assertion that pins the value. It also explains why extreme mutation is a productive detector even though the constant regime is a weak certifier (Corollary 4.3): the two roles are different, and the surfaced gap is the intent test the suite is missing.

Proposition 9.4b (the mixed ceiling — the score a tool actually reports). Definition 9.2 discounts ω -equivalent mutants, so the absolute ω -score of the paragraph above is not the number a tool reports: production tools (Descartes, PIT) count a mutant as non-equivalent when it changes the returned value — the exact oracle — and as killed when the suite notices — the weak oracle ω — which is exactly the mixed reading Proposition 9.4 adopts. Applied to the ceiling, that reading gives a strictly stronger theorem. The absolute mixed score (exact-oracle non-equivalence, ω -kills) is 1 iff

$$O = I \quad \text{and} \quad \omega^{-1}(\omega(b)) = \{b\} \quad \text{for every } b \in I,$$

i.e. the suite observes every reachable output and the oracle separates every reachable value from every other. Under the exact oracle the second clause is automatic ($\omega = \text{id}$ is injective) and this collapses to Theorem 4.5.

Proof. ($\Leftarrow$) Let p be non-equivalent under the exact oracle: some $b \in \text{Mov}(p) \cap I$. Then $b \in O = I$ and $p(b) \neq b$, and $\omega^{-1}(\omega(b)) = \{b\}$ forces $\omega(p(b)) \neq \omega(b)$, so T ω -kills p at b ; hence every exact-non-equivalent mutant is ω -killed. ($\Rightarrow$) If $b \in I \setminus O$, the guard γ_b is exact-non-equivalent and ω -unkilled (its only moved value b is unobserved), so it survives. If some $b' \neq b$ shares b 's ω -class with $b \in I$, the operator sending $b \mapsto b'$ is exact-non-equivalent yet has $\text{Det}_\omega = \emptyset$, so it survives. Either failure drops the score below one. ■

A. Empirical: an illustration of oracle-relativity

Proposition 9.4 says whether a covered method is pseudo-tested depends on the oracle. We illustrate the magnitude of that dependence — not confirm the biconditional, which no aggregate of this shape can, for the reason in the threats below — with two oracles ordered by kill strength: a value oracle, under which a kill requires a test assertion to distinguish the returned value, and a crash-inclusive oracle, under which a downstream exception or timeout also counts as a kill. We ran the constant perturbation return 0 at each return site against the existing suites of five widely-used Python libraries, over a seeded random sample of public functions and methods; a method enters the denominator only when a covering test executes the mutated return, and counts as pseudo-tested under an oracle iff no covered return is killed by it.

Across the 122 covered functions and methods of the four measurable projects, 43% are pseudo-tested under the value oracle — within the 1–46% [15] () measured across 21 Java projects — but only 6% under the crash-inclusive oracle. The 37-point gap is the population of constant

TABLE I
Mechanization ledger: paper results and their Lean names.

Project	Covered w/ const	Value oracle	Crash oracle
boltons	27	63%	4%
inflect	33	52%	9%
toolz	46	33%	0%
humanize	16	19%	19%
click (excl.)	2	—	—
Pooled	122	43%	6%

returns a traditional mutation score counts as killed (a downstream crash caught them) while the returned value is unpinned: the same code is roughly seven times more pseudo-tested once crash detection is discounted. That gap is the empirical shadow of §9’s oracle-relativity.

Threats to validity — this is an illustration, not a test of Proposition 9.4. (i) The direction is tautological: crash-inclusive kills are a superset of value kills, so the crash-inclusive pseudo-tested set is a subset of the value one and $6\% \leq 43\%$ holds by construction — only the magnitude of the gap carries information, not its sign, and a per-method biconditional cannot be confirmed by two monotone aggregates. (ii) The operator is return 0 at every site, which raises `TypeError` downstream at `str/list/object` return sites; those crashes are type artifacts rather than genuine crash-detection, so the crash column is not comparable across sites (Descartes uses a type-appropriate family — `null / "" / 0 / false`). (iii) return 0 at each return site is a per-return-site mutation, which Remark 2.6 excludes: the theory characterizes the uniform post-composition $p \circ f$, so this measures a related but distinct operator. (iv) A value-pinning `==` assertion is close to the exact oracle $\omega = \text{id}$, but a property-assertion (a field, a length, a type) is a strictly weaker ω (Definition 9.1), so “value oracle” names a family of oracles and the column mixes them. A measurement that actually tested Proposition 9.4 would inspect, per surviving mutant, which ω -class its assertions induce and check the mixed condition directly — future work. (click is excluded as a harness artifact — its command/context objects do not profile in process — not a coverage limitation.)

X. The intent residual

The results assemble into a single profile of what is knowable about a function’s output behavior, and where that knowledge ends.

Mechanically knowable, and automatable. Under an oracle that asserts something, a suitable mutation regime determines the distribution of a function’s outputs up to an exact ceiling: which distinct values are reachable, and whether the suite has observed each (Theorem 4.5). The minimal regime that makes this exact is the value guards (Corollary 4.1); the knowledge is achievable exactly on finite return types (Corollary 4.2), decidable exactly when footprint containment is (Theorem 5.2), program-independent (Proposition 7.2), and priced at $|f(D)|$ tests

(Theorem 8.1). None of this requires knowing what the program is for; it is read off the program and its executions, and a tool can compute it. Strictly, a tool never enumerates I : it observes O and the unkillable guards, and an unkillable guard γ_b certifies $b \in I \setminus O$ (a reachable output no test observed). A full score is therefore read as “no guard left unkillable” — $I \setminus O = \emptyset$ — not as a direct comparison to a reachable set the tool cannot see; the ceiling’s content is exactly that this indirect certificate equals output coverage.

The residual, requiring a human intent test. Two things the regime provably cannot determine from the program alone. First, structurally: faults that separate two inputs mapped to the same output are outside the technique’s reach entirely (Remark 4.6). Second, and centrally: which of the output distinctions the regime certifies actually matter. The oracle-relative theory (§9) makes this exact — the gap $\text{Mov}(p) \setminus \text{Det}_\omega(p)$ is a real non-equivalence the mechanical layer produces and no assertion pins, and closing it is not more mutation but a human supplying the missing assertion: an intent test stating that on this input the returned value is what it should be. A pseudo-tested method is precisely a method whose output behavior the regime has fully mapped and whose intent no test has fixed. The knowability boundary is therefore not between easy and hard, nor cheap and expensive; it is between output behavior — determined mechanically, up to the ceiling — and intent — never committed to the program, and supplied only by a teacher. The full profile of the first is this paper’s theorem; the finiteness and namedness of the second is what makes the division a workflow rather than a fog.

XI. Related work, scope, and mechanization

Related work. The claim that mutation testing “lacks a completeness theorem” needs a precise qualifier. There is no program- and suite-independent characterization of adequacy against the semantic-subsumption order — that order is undecidable ([9]), and the classical constructive results are relative by construction (sufficient operators, experimental, [13]; minimal / dominator mutant sets, test-set- and pool-relative, [1]; [2] give undecidability of related decision problems). But program-independent sufficiency results do exist and this paper must situate against them: the minimal sufficient relational- and conditional-operator

sets ([16]; [8]) prove that a few mutants dominate the rest at the point of mutation, for any predicate in any program — exactly the kind of program-independent statement the present theory generalizes for the output-operator class, and the parallel is worth drawing rather than eliding. Likewise the coupling effect is not merely assumed: Wah’s [17] analyses establish it under restricted fault models, so the honest statement is “unproven in general,” not “never proven.” Two of this paper’s own constructs are prior art under other names: Theorem 4.5’s ceiling — a full score means every reachable output was observed — is the output-uniqueness criterion of [18], reached here as a theorem about footprints rather than posited as a coverage criterion; and the value/oracle gap of §9 is checked coverage ([19]), the existing dynamic measure of whether an assertion actually constrains a computed value. Extreme mutation and pseudo-testedness are [11] and [15] (, Descartes); we recover extreme mutation as the $n = 2$ optimum (Corollary 4.3, with the oracle-separation caveat of Remark 9.3b) and characterize pseudo-testedness in the model’s own terms (Proposition 9.4). The teaching-dimension reading is [5].

New here, and stated against that prior art rather than as if the neighborhood were empty: the program- and suite-independent notion of sufficiency for an arbitrary target class (Definition 3.1) and its exact footprint characterization over the whole output-operator lattice (Theorems 3.2 / 3.2b) — a generalization of the point-of-mutation dominance results above from specific operator schemas to the full class; the value-guard basis and its size (Corollary 4.1); the detection factoring that explains why the theory is program-independent where program-text mutation is not (Proposition 7.2); and the mixed-oracle characterization of the intent residual (§9–§10). Proposition 6.1 (coupling not entailed for output mutation) is a clarification of Remark 3.3, not an independent contribution. A companion paper (Effect and Meaning in Mutation-Based Specification) develops the effect/meaning frame and the decidable-coupling result this paper’s residual points to; the two-sign specification-complexity paper develops the teacher-supplied negative sign.

Scope. The theory concerns output/extreme mutation of pure total functions; §§2–8 assume an exact oracle, §9 removes it. It does not extend to general program-text mutation, and makes no claim about the coupling hypothesis for program-text faults. The effect layer has its own undecidability — general footprint containment (Theorem 5.2(3)) — which we characterize, not dissolve. The central quantitative result (a full score equals output coverage) is deliberately modest: it says what a score means, not that a tool should compute it in preference to measuring output coverage directly.

Mechanization. The formal core (§§2–8) and the oracle-relative core of §9 are machine-checked in Lean 4 / Mathlib (proofs/adequacy_completeness.lean), stated di-

rectly over programs $f : D \rightarrow R$ and finite suites $T : \text{Finset } D$ and bridged to the footprint level, every carried theorem `#print axioms-clean ([propext, Classical.choice, Quot.sound]`, no `sorryAx`) — including the corrected mixed-oracle Proposition 9.4 (`pseudo_tested_mixed_iff`), noted after the table.

Proposition 9.4 is carried in both forms. `pseudo_tested_mixed_iff` is the corrected mixed-oracle statement the prose states — non-equivalence under the exact oracle ($\text{Mov}(\text{‘return } c\text{’)} \cap I \neq \emptyset$, i.e. $I \neq \{c\}$) together with ω -survival on the observed set ($\text{Det}_\omega(\text{‘return } c\text{’)} \cap O = \emptyset$, i.e. $O \subseteq \omega^{-1}(\omega(c))$) — which correctly includes the paradigm ω -equivalent pseudo-tested method; `pseudo_tested_iff` retains the narrower ω -non-equivalent biconditional. Both are machine-checked by automated proof search (no remaining `sorry`; the verified build reports axioms `[propext, Classical.choice, Quot.sound]`).

Two results are not carried by the machine as single theorems. First, Theorem 5.2’s decidability spectrum — a meta-statement whose load-bearing reduction to footprint containment is the checked Theorem 3.2. Second, Remark 9.3b’s constant-under-weak-oracle claim rests on `detW_const` but its full “iff $n = 2$ and ω separates” form is not carried as a single theorem. To reproduce: place the .lean file in a Lean 4 project with Mathlib (lake exe cache get) and check e.g. `#print axioms pseudo_tested_mixed_iff`.

XII. Conclusion

For one operator class we replace a heuristic — a high score read as adequacy on the strength of the coupling hypothesis — with a theorem about what the score determines. The mechanically knowable profile of a function’s output behavior is exactly characterized: which regime suffices to certify it (Theorem 3.2), the minimal such regime (Corollary 4.1), the exact content of a full score (Theorem 4.5, output coverage), where that knowledge is achievable (finite types) and decidable (footprint containment), and what it costs ($|f(D)|$ tests). What the regime cannot determine — which of those certified distinctions bear on the program’s intent — is the residual a weak oracle exposes and a human closes with an intent test. A mutation score, for this class, is thus not a quantity trusted empirically but an object with a characterized semantics, and the line between what a tool can know about a function and what only its author can say is drawn exactly, and mechanically checked to the kernel.

TABLE II
Mechanization ledger: paper results and their Lean names.

Result	Lean name	Status
Prop 2.4 (operator $\equiv$ footprint, over programs)	progScore_iff_scoreAt	mechanised
Remark 2.5 (every subset is a footprint)	mov_surjective, mov_image_univ	mechanised
Thm 3.2 (characterization)	footprint_characterization	mechanised
Thm 3.2 (over programs)	progComplete_characterization	mechanised
Thm 3.2 (general II)	footprint_characterization_general	mechanised
Cor 4.1 (value-guard basis)	absolute_iff_guards	mechanised
Cor 4.2 (finite/infinite dichotomy)	progComplete_univ_infinite	mechanised
Cor 4.3 (constants iff $n=2$)	constants_iff_card_two	mechanised
Thm 4.5 (ceiling)	ceiling	mechanised
Prop 6.1 (coupling not entailed)	coupling_fails	mechanised
Prop 7.1 (subsumption = containment)	subsumes_iff_subset	mechanised
Prop 7.2 (detection factors)	det_factors	mechanised (rfl)
Thm 8.1 (min certifying suite)	certify_lb, certify_ub, certify_infinite	mechanised (observed-set core)
Bridge (realizability, $\text{program} \leftrightarrow \text{footprint}$)	realizable, progComplete_iff_complete	mechanised
Prop 9.3 (oracle-relative core)	detW_singleton_of_nonconstant, detW_const	mechanised
Prop 9.4 (pseudo-testedness, mixed oracle)	pseudo_tested_mixed_iff	mechanised
Prop 9.4 (narrow ω -non-equivalent form)	pseudo_tested_iff	mechanised

References

- [1] Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: ICST (2014)
- [2] Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. *Acta Informatica* 18(1), 31–45 (1982)
- [3] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *IEEE Computer* 11(4), 34–41 (1978)
- [4] de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: CADE (2021)
- [5] Goldman, S.A., Kearns, M.J.: On the complexity of teaching. *J. Comput. Syst. Sci.* 50(1), 20–31 (1995)
- [6] Hamlet, R.G.: Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* SE-3(4), 279–290 (1977)
- [7] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* 37(5), 649–678 (2011)
- [8] Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: FSE (2014)
- [9] Kurtz, B., Ammann, P., Offutt, J.: Static analysis of mutant subsumption. In: ICST Workshops (Mutation), pp. 1–10 (2015). <https://doi.org/10.1109/ICSTW.2015.7107454>
- [10] The mathlib Community: The Lean mathematical library. In: CPP (2020)
- [11] Niedermayr, R., Juergens, E., Wagner, S.: Will my tests tell me if I break this code? In: CSED@ICSE, pp. 23–29 (2016). <https://doi.org/10.1145/2896941.2896944>
- [12] Offutt, A.J.: Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.* 1(1), 5–20 (1992)
- [13] Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5(2), 99–118 (1996)
- [14] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. *Advances in Computers* 112, 275–378 (2019)
- [15] Vera-Pérez, O.L., Danglot, B., Monperrus, M., Baudry, B.: A comprehensive study of pseudo-tested methods. *Empirical Software Engineering* 23(3), 1908–1935 (2018)
- [16] Kaminski, G., Ammann, P., Offutt, J.: Improving logic-based testing. *Journal of Systems and Software* 86(8), 2002–2012 (2013)
- [17] Wah, K.S.H.T.: An analysis of the coupling effect I: single test data. *Science of Computer Programming* 48(2–3), 119–161 (2003)
- [18] Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: ISSA 2014, pp. 181–192 (2014)
- [19] Schuler, D., Zeller, A.: Checked coverage: an indicator for test quality. *Software Testing, Verification and Reliability* 23(7), 531–551 (2013)