

HyBind: Structural Analysis of PQ/T Hybrid Cryptographic Usage

Jongbeom Ahn¹[0009-0005-7205-8409], Minseo Kim¹[0009-0008-1979-5778],
Moonsun Heo¹[0009-0005-6631-199X], Subeen Cho¹[0009-0004-4352-8021], and
Hwajeong Seo¹[0000-0003-0069-9061]

¹Department of Convergence Security,
Hansung University, Seoul (02876), South Korea,
{ajb11191128, msqoo0212, ky060261, chosubin1208, hwajeong84}@gmail.com

Abstract. The quantum threat to traditional public-key cryptography makes migration to post-quantum cryptography necessary. During migration, the presence of both post-quantum and traditional (PQ/T) algorithms does not establish that their outputs jointly determine a key or verification decision. We propose HyBind, an intraprocedural Python source-analysis framework using registered API contracts. HyBind traces component secrets to returned keys and checks whether signature acceptance requires both verifiers to succeed. Function-return and execution-effect checks account for alternative outcomes and changes to local bindings. Two internal presence-based heuristics give identical decisions for joint versus omitted secret contribution and AND versus OR verification, while HyBind distinguishes these structures. All 50 controlled fixtures match their specified outputs with the annotation assumption enabled. A separate suite of 33 programs, including 15 using native cryptographic APIs, matches its specified positive-finding counts in both settings. Native runtime checks exercise ML-KEM/X25519 key derivation and ML-DSA/Ed25519 acceptance. Relaxing return or effect requirements admits traditional-only key results as hybrid findings in the evaluated cases. HyBind provides the contributing calls and outcome evidence needed to review whether an individual traditional invocation participates in a PQ/T hybrid.

Keywords: Post-quantum cryptography · PQ/T hybrid cryptography
· Static analysis · Cryptographic migration · Python

1 Introduction

Migration to post-quantum cryptography requires identifying where and how quantum-vulnerable public-key algorithms are used [8, 9]. A traditional call can remain as a component of a PQ/T hybrid [4]. An inventory identifies the algorithms present, but migration review must also determine whether their outputs contribute to a joint result. For key establishment this means tracing secrets to a

returned key; for signatures it means checking the verifier successes required for acceptance. Listing 1 isolates the first question with different combiner inputs; `ct` is the ciphertext.

Listing 1. Same producer calls, different secret contributions.

```

1 # Imports from crypto_stubs are omitted.
2 def joint_key(pq_pk, trad_sk, peer_pk):
3     ct, pq = mlkem_encapsulate(pq_pk)
4     trad = x25519_exchange(trad_sk, peer_pk)
5     return derive_session_key(hybrid_combiner(pq, trad))
6
7 def omitted_secret(pq_pk, trad_sk, peer_pk):
8     ct, pq = mlkem_encapsulate(pq_pk)
9     trad = x25519_exchange(trad_sk, peer_pk)
10    return derive_session_key(hybrid_combiner(trad))

```

Only `joint_key` uses both secrets in its combined result. Even that local combination would be insufficient if another branch returned a traditional-only key. For signatures, a similar distinction arises when both verifiers are called but either result can permit acceptance. These cases require checking how component calls affect the enclosing function’s result.

Cryptoscope recovers operation-level cryptographic assets and their context [7], and cryptographic bills of materials can represent hybrid dependencies [5]. HyBind supplies source-level criteria for deciding when a traditional invocation can be classified as part of such a hybrid. It combines local contribution checks with summaries of the enclosing function’s returns. Registered contracts identify the roles and success conventions of component APIs; binding and execution-effect checks account for Python behavior that can change the calls or their results. Our contributions are:

1. **A structural classification model.** We separate checks on component inputs from checks on the enclosing function’s return outcomes. Within the modeled key outcomes, the key criteria exclude traditional-only returns; the signature criteria require acceptance to depend on qualifying verifier pairs and link each reported pair to its own return evidence.
2. **An intraprocedural Python analyzer.** HyBind combines exact-symbol API recognition, local value flow, return summaries, and execution-effect checks under registered API contracts, including native PQ APIs. Each finding records its evidence or unresolved requirements.
3. **An evaluation of these criteria.** Controlled comparisons show that HyBind distinguishes secret omission and OR acceptance where two internal presence heuristics retain identical decisions. Two suites of 50 and 33 author-defined programs test specification agreement and native API support. Runtime checks and removal experiments connect the return and effect checks to the traditional-only outcomes they exclude.

2 Problem Setting and Scope

2.1 What HyBind Classifies

Calls to the same algorithms can appear in programs with different combiner inputs or final acceptance conditions. Assignments, branches, and overridden returns further separate a call’s presence from its role in the result. HyBind therefore distinguishes invocation sites, candidate relationships, and reported findings.

A scan analyzes the Python files under one selected root. A modeled operation is an API invocation or a supported byte-concatenation expression, identified by its file, lexical scope and source position. A candidate is a combination operation or a pair of verifier calls. A finding reports a classification, locations, and associated component calls; several findings can occur in one scan.

Scopes distinguish the module, class bodies, and function bodies, including nested functions, using their lexical nesting and definition positions. The return checks summarize each function body separately; f denotes the enclosing function scope. Each reported relationship is tied to local values and exits that a reviewer can inspect.

HyBind recognizes cryptographic calls, constructs combiner and verifier-pair candidates, classifies their structure, and then applies the migration policy. A *positive finding* records a structurally valid key-establishment or signature-verification candidate: it passes its local checks together with separately applied function-level return and execution-effect checks. Section 3.5 maps this outcome to the exact output fields. Table 1 summarizes the reported classifications.

Table 1. Overview of reported source classifications.

Reported classification	Source-level meaning
Traditional use	A traditional invocation without a PQ/T relationship established by the analysis.
Unbound coexistence	Both algorithm families occur, but no joint-result relationship is established.
Structurally valid candidate	The candidate passes its local and enclosing-function checks.
Invalid hybrid relationship	A required contribution, provenance, or acceptance condition is violated.
Unresolved candidate	Required provenance or analysis evidence remains unresolved.

These classifications describe findings and may coexist in one scan. Section 4.1 gives the migration policy applied to them.

We use PQ/T terminology from RFC 9794 [4]. In multi-algorithm key establishment, a combiner can process component shared secrets into a common

result [1]. The examples use contracts for ML-KEM key encapsulation and ML-DSA signatures, standardized in FIPS 203 and FIPS 204, respectively [10, 11]. The signature profile considered here requires both component verifiers to succeed; profiles permitting single-component acceptance are outside this classification policy.

2.2 Trust Boundary and Assumptions

The analyzer takes Python source, registered API contracts, version-pinned algorithm metadata, and a migration policy. Its source model follows Python 3.12. The runtime and registered library semantics are trusted: contracts specify operation roles, argument positions, output values, and whether successful verification returns a Boolean or completes without raising a specified exception.

Under these assumptions, HyBind classifies candidates using component inputs and enclosing-function return outcomes. This targets omitted secret contributions, single-verifier acceptance, message-origin mismatches, and traditional-only fallback. Binding and effect checks withhold positive findings when operations such as attribute access or truth-value tests may invoke unresolved user-defined behavior [16].

The default mode does not use primitive parameter annotations to rule out runtime dispatch, consistent with the non-enforcing role of type hints [18]. The optional annotation assumption treats parameters annotated with `bool`, `int`, `float`, `complex`, `str`, or `bytes` as having built-in-like behavior without user-defined protocol dispatch. This trusted premise is stronger than ordinary subtype-compatible typing. Appendix A.5 distinguishes the facts supplied by annotations from receiver-type recognition.

3 Operation-Level Structural Analysis

Local data flow identifies which component outputs form a candidate. HyBind then checks the enclosing function’s returns to account for alternative branches and overridden results. Key summaries track the secrets in returned values; signature summaries track the verifier successes required for acceptance. Binding and execution-effect checks determine whether these summaries have the information they require. Figure 1 shows the analysis. Examples below use the default mode and evaluation policy.

3.1 API Recognition

HyBind resolves imports, receiver types and local bindings, then matches calls against registered API symbols and argument contracts. Output roles distinguish a shared secret from a ciphertext or an encapsulation tuple. Rebinding invalidates the associated API facts.

Table 2 shows representative fixture contracts. Their `crypto_stubs` implementations use fixed bytes and simple operations. Native contracts cover ML-KEM, ML-DSA and traditional methods from `cryptography`, together with

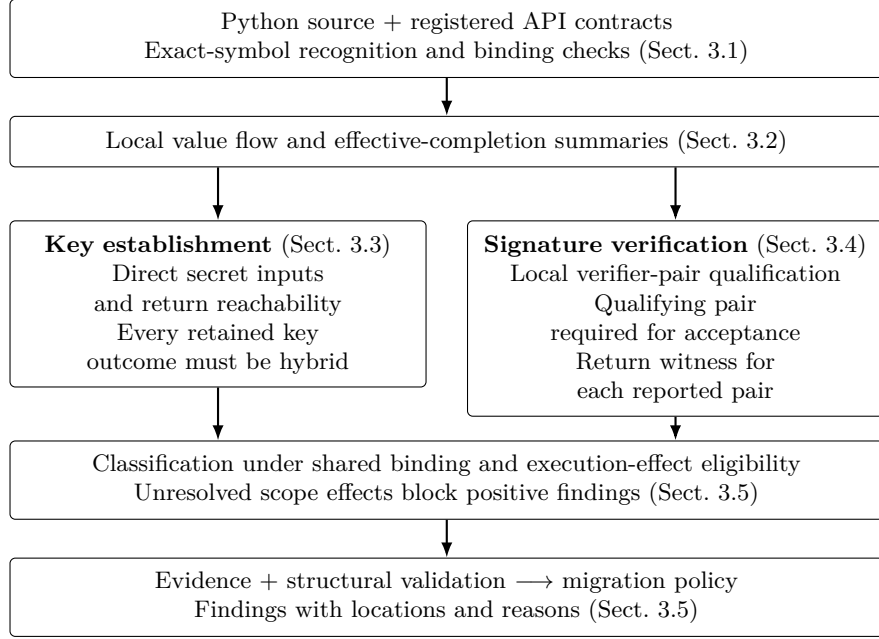


Fig. 1. Evidence and checks used for structural classification. Each positive signature pair has its own accepting-completion record.

a bounded HKDF-SHA256 profile. Exact bytes addition preserves component contributions through local assembly steps. In HKDF derivation, only the key-material argument supplies secret inputs. Appendix A.7 gives the supported configurations and binding rules.

Table 2. Representative exact-symbol contracts in `crypto_stubs`.

API / variant	Arguments (in order)	Output or success convention
<code>mlkem_encapsulate</code>	public key (exactly 1)	Tuple: ciphertext, shared secret
<code>x25519_exchange</code>	private key, peer public key (2)	Shared secret
<code>hybrid_combiner</code>	One or more inputs	Combined value
<code>derive_session_key</code>	combined value (1)	Derived key
<code>ml_dsa_verify</code> , <code>ecdsa_verify</code>	signature, message, public key (3)	Boolean success
Exception-raising verifiers	signature, message, public key (3)	Successful completion or modeled invalid-signature exception

Contracts also specify success conventions. ECDSA verification returns `None` on success and raises `InvalidSignature` on failure [12]; X25519 exchange returns shared-secret bytes [15]. HyBind interprets these contracts statically and checks operand-evaluation effects separately.

3.2 Local Values and Function Exits

Assignments use versioned local symbols to distinguish successive bindings. The analyzer follows aliases and retains incoming sources in a *phi-like merge record* at supported branch joins. An evidence graph connects primitive calls, secrets, derived values, combiners, messages, verification patterns, and returns through production, combination, and flow edges.

Effective completions. A *completion* is an analysis record describing how a path leaves a block or function, together with its returned value and tracked state. It becomes an *effective completion* after handlers and any `finally` block have been accounted for. Thus, a return expression and the eventual function result can differ.

These summaries use transfer rules over Python’s abstract syntax tree (AST) for sequencing, branches, handlers, and selected loops. Throughout this paper, “all completions” refers to the exits retained by the analysis under its transfer rules and execution-effect checks.

3.3 Key-Establishment Checks

Applying the key checks. In Listing 1, the output-role contract identifies `pq` as the shared secret and `ct` as the ciphertext. HyBind reports a positive finding for `joint_key`.

In the omitted-secret alternative of Listing 1, the PQ secret does not reach the candidate combiner, producing an invalid finding. A separate return classified as traditional-only by the key-return analysis also blocks a positive finding when a hybrid-key return is present elsewhere.

A positive key finding requires shared-secret inputs from both families and an output that reaches a return expression. The enclosing function must have at least one key-relevant normal completion, and every such completion must produce a supported hybrid key. The latter check is shared by candidates in the same function.

A *directly bound secret* is a shared-secret graph node resolved from one of the call’s arguments, including supported aliases, nested secret producers, and exact bytes assembly. Other combiner/KDF outputs remain distinct derived values. A candidate must have at least one directly bound secret, or two for a KDF. In Listing 1, the combiner is the candidate and the one-input KDF propagates its result to the return.

Value summaries for key establishment. Each value summary records possible PQ, traditional, and combiner producers, together with its value kind, hybrid status, and unresolved information. A secret-producing call introduces its identifier. A combiner collects its inputs' dependencies, and a KDF propagates them to a key. Assignments transfer summaries to new bindings. At a branch merge, producer sets are unioned, while hybrid status requires every incoming alternative to be hybrid. Unresolved alternatives, incompatible value kinds or differing algorithm families make the merged value unresolved. The analysis also tracks whether each path state uses supported transfers.

For key establishment, normal results are classified as hybrid, traditional-only, PQ-only, unresolved, or reached through an unsupported transfer. Boolean **and/or** preserves possible returned operands and follows literal short-circuit decisions, including conditional assignments. Skipped operands leave the key state unchanged. Supported bytes addition retains both inputs; other binary/unary operations and rich comparisons retain operand dependencies as unresolved values.

The explicit **non_key** kind covers registered encryption/decryption, signing, and verification outputs and Boolean results from **not**, identity, and membership checks. These outcomes and escaping exceptions are excluded from the key-return obligation. Unclassified literals, containers, missing values, and fall-through remain unresolved. Execution effects are checked separately, including effects of evaluating non-key expressions. Consequently, registered decryption outputs are not examined as possible transported keys: a path returning an RSA-decrypted value does not by itself block a positive hybrid finding elsewhere in the function, although the RSA invocation remains a separate traditional-use finding with policy state **FAIL**.

Let b be a candidate combiner in enclosing function scope f ; subscripts K and S distinguish key and signature checks. Write P_b and T_b for the PQ and traditional shared secrets directly bound to b , $R(b)$ for reachability from its output to a return expression, and $\text{Eligible}(f)$ for satisfaction of the function's binding and execution-effect checks. Reachability uses lexical value flow and producer identifiers retained by proven key completions. Let $\mathcal{C}_K(f)$ contain the key-relevant normal completions; escaping exceptions and outcomes assigned **non_key** by the value-summary rules are excluded.

$\text{Supported}(c)$ means that completion c uses supported transfers. The predicate $\text{HybridKey}(c)$ means that c is classified as a hybrid key.

Every positive structural key-establishment finding satisfies the following output invariant:

$$\begin{aligned} \text{Positive}_K(b, f) \implies & \text{Eligible}(f) \wedge (P_b \neq \emptyset) \wedge (T_b \neq \emptyset) \\ & \wedge R(b) \wedge (\mathcal{C}_K(f) \neq \emptyset) \\ & \wedge \bigwedge_{c \in \mathcal{C}_K(f)} (\text{Supported}(c) \wedge \text{HybridKey}(c)). \end{aligned} \tag{1}$$

Different branches may use different hybrid combiners. Component associations are specific to invocation identifiers; traditional-only outcomes remain subject

to the migration policy. Table 3 summarizes the decisions, with scope eligibility taking precedence over local checks.

3.4 Signature-Verification Checks

The acceptance operator and API contract matter. Two verifier calls are insufficient when the acceptance condition requires only one of them.

Listing 2. Boolean-returning component verifiers.

```

1 from crypto_stubs import ml_dsa_verify, ecdsa_verify
2
3 def verify(msg, pq_sig, trad_sig, pq_pk, trad_pk):
4     return (ml_dsa_verify(pq_sig, msg, pq_pk)
5           and ecdsa_verify(trad_sig, msg, trad_pk))

```

Both verifiers in Listing 2 have Boolean-return contracts. The message input has the same singleton provenance at both calls, while the signature inputs have distinct local sources. A truthful result of the conjunction requires both component results. HyBind reports a structurally valid finding. Replacing **and** with **or** preserves the calls and their message relationships, but permits acceptance after only one successful component. The OR fixture receives an invalid finding under the configured AND profile.

Success conventions also affect this pattern. The registered ECDSA verifier in **cryptography** returns **None** on success and raises **InvalidSignature** on verification failure. An **and** expression with these results does not implement joint verification success and receives an unresolved finding. Exception-based verification must instead connect successful completion of both calls to acceptance.

A candidate verifier pair $p = (v_P, v_T)$ must match a supported pattern requiring both verifiers to succeed. Boolean composition and guards use Boolean-returning verifiers; exception patterns use the raising convention. The calls must be distinct, belong to the required algorithm families and be retained by the frontend’s reachability processing. $\text{Pair}(p)$ denotes these pattern and API-contract requirements together with the following provenance conditions.

Operand provenance. HyBind compares verifier operands by their resolved local origins. An operand is *admissible* when its origin set is nonempty and resolved, and its expression is a local name or a supported immutable literal. Names resolve through aliases and merge records; literals have value-based origins. Both message operands must be admissible and resolve to the same single origin. The signature operands must be admissible and have disjoint origin sets, which need not be singletons. These checks concern source origins, not equality or inequality of runtime byte values. In Listing 2, both messages resolve to **msg**, while **pq_sig** and **trad_sig** supply separate signature origins. An attribute such as **box.message** has unresolved provenance because lookup may invoke a descriptor.

Function-level return check. The return-decision model treats truthful function results as acceptance and known falsey results or supported fall-through as rejection. For exception-style component APIs, the supported pattern links successful completion of both verifiers to a truthful return, with rejecting handlers.

For signatures, the return summary distinguishes four alternatives: (1) a decision carrying a qualifying verifier pair's dependency; (2) a value known to be truthful without that dependency; (3) a known falsey result, including **False**, **None**, or zero; and (4) an unresolved decision. The second alternative is an accepting bypass. A pair-dependent decision records the calls required for joint success, regardless of whether their runtime outcomes are known. Escaping exceptions supply no accepting return.

A positive signature finding requires supported rejection or a qualifying pair dependency at every retained return decision. A function may also contain pairs whose results are discarded, so each reported pair needs its own return evidence: at least one retained decision must carry that pair's two verifier-call identifiers. Different branches may use different qualifying pairs.

Let $\mathcal{C}_S(f)$ be the effective return decisions retained by the analysis. $\text{Reject}(c)$ denotes a supported rejection. $\text{Justified}(c)$ denotes a decision with a recorded dependency on a qualifying verifier pair: $\exists p' [\text{Pair}(p') \wedge \text{Requires}(c, p')]$. Every positive structural signature finding satisfies the output invariant

$$\begin{aligned} \text{Positive}_S(p, f) \implies & \text{Eligible}(f) \wedge \text{Pair}(p) \\ & \wedge \exists c \in \mathcal{C}_S(f) : \text{Requires}(c, p) \\ & \wedge \bigwedge_{c \in \mathcal{C}_S(f)} (\text{Reject}(c) \vee \text{Justified}(c)). \end{aligned} \tag{2}$$

Here Requires denotes a recorded decision dependency. A candidate witness records the return location and its two verifier-call identifiers. Both invariants express necessary conditions over the completions retained by the analysis.

Table 3. Principal structural decisions, grouped into common, key, and signature checks. Migration-policy outcomes are assigned separately (Sect. 4.1).

Candidate	Condition	Validation state
Either	Scope eligibility or required component evidence unresolved	UNKNOWN
Key	Both families occur, but the combination omits one required secret	INVALID
	Both secrets reach the combiner, but return reachability or the hybrid-key completion condition is unresolved	UNKNOWN
Signature	Local pattern permits either verifier alone under the AND profile	INVALID
	API/control convention, signature separation, or message origin unresolved	UNKNOWN
	Resolved unequal message-origin sets or an accepting bypass	INVALID
	Return decision or candidate accepting witness unresolved	UNKNOWN
Either	All operation-specific checks and the record consistency check pass	STRUCTURALLY_VALID

3.5 Execution-Effect Checks and Reporting

A completion override blocks a positive finding. A return expression need not determine the value that leaves a function.

Listing 3 uses the imports of Listing 2.

Listing 3. A return in finally overrides the verification result.

```

1 def verify(msg, pq_sig, trad_sig, pq_pk, trad_pk):
2     try:
3         return (ml_dsa_verify(pq_sig, msg, pq_pk)
4                 and ecdsa_verify(trad_sig, msg, trad_pk))
5     finally:
6         return True

```

The **finally** block replaces the pending verification result with **True**. Hy-Bind reports an unresolved finding for this case and emits an application-code diagnostic.

Execution-effect checks account for binding changes, implicit user-code calls and completion overrides under Python 3.12 semantics [16]. An unresolved effect blocks positive findings in the affected lexical scope. Appendices A.1 and A.5–A.6 specify the supported operations, exception treatment and completion rules.

Before reporting a positive finding, a common consistency check verifies that its effect records are complete and its evidence requirements are resolved; otherwise the finding becomes UNKNOWN (Appendix A.6).

Output dimensions. Evidence, validation, and policy are separate finding fields (Table 4). E0 denotes indirect or diagnostic evidence; E1 identifies an algorithm invocation; E2 records a candidate or an insufficient relationship; and E3 records a structural determination. E4 is reserved for execution-linked or cryptographic validation and is not produced by source analysis. E3 can accompany an invalid structure, while an UNKNOWN candidate can receive policy state FAIL. A *positive finding* means an E3 key-establishment or signature-verification finding with validation state STRUCTURALLY_VALID, independently of policy. The predicates Positive_K and Positive_S refer to this meaning.

The JSON field `analysis_complete` records application-code diagnostic status. The field is true when no diagnostic concerns a file classified as application code. Appendix A.3 defines the scan-root-relative path convention.

Table 4. Finding dimensions and scan metadata.

Axis	Values	Meaning
Evidence	E0–E4	Kind of available evidence
Validation	STRUCTURALLY_VALID, INVALID, UNKNOWN, CLAIMED_NOT_PROVEN	Structural status of the reported use or relationship; the last denotes a claim without binding proof
Policy	PASS, WARN, FAIL	Consequence under the selected migration policy
Scan	<code>analysis_complete:</code> <code>true/false</code>	True when no diagnostic concerns files treated as application code

Under the evaluation policy (Sect. 4.1), a structurally valid hybrid candidate has E3 evidence, STRUCTURALLY_VALID status, and policy state PASS. For a traditional-use finding, STRUCTURALLY_VALID means that the traditional invocation is established; it does not denote a valid hybrid. A missing PQ secret has E3/INVALID/FAIL. An unresolved execution effect can yield E2/UNKNOWN/FAIL and an application-code diagnostic. The artifact exports JSON, Markdown and SARIF reports with evidence and source locations.

Recorded explanations. These default-mode JSON excerpts illustrate a valid verifier pair and an unresolved completion override. Locations refer to the original fixtures; listings abbreviate their source. The valid-pair record also contains `accepting_witnesses`, which links the accepting return to the two verifier-call identifiers. This field makes the candidate-specific requirement in Eq. (2) inspectable.

```

hybrid_sig_and/app.py:13; CRYPTO-HYB-008
evidence_level=E3; validation_state=STRUCTURALLY_VALID
same_message=true; distinct_signature_components=true
completion_gate=proven; unsupported_effects=[]
v7_finally_signature_override/app.py:5; CRYPTO-HYB-010
evidence_level=E2; validation_state=UNKNOWN
unsupported_effects: finally block at line 7 may override a
cryptographic return or verification decision; E3 is withheld

```

4 Evaluation

We first compare HyBind with presence-based decision rules to test whether contribution and acceptance checks change the classification of traditional calls. Two controlled suites then test specification agreement, sensitivity to annotation assumptions and native API support. Finally, removing selected checks shows which return and effect information is needed for the reported classifications.

4.1 Protocol and Measurement Units

Controlled inputs. The conformance suite contains 50 author-defined fixtures covering joint-result patterns, structural violations, and analysis boundaries. They vary secret contribution, acceptance operator, message origin, API success convention, return completion, and execution effects.

`experiments/ground_truth/source.yaml` records expected findings, policy exits, and application-code diagnostic status. The scanner receives API contracts and the selected annotation assumptions; the evaluator compares its outputs with the specifications. A fixture matches when all expected findings are present, no unexpected findings appear, and its exit and diagnostic status agree. Some fixtures contain multiple operations.

Measurement units. Specification agreement is counted per fixture, including specifications that expect invalid or unresolved findings. A scan may report several findings, each with its own policy state. We separately count scans with no application-code diagnostic and scans containing an UNKNOWN finding. Positive findings follow Sect. 3.5. Each fixture is counted once per predicate; finding-level categories can overlap within a scan.

Evaluation policy and scan status. All source evaluations use the following migration policy: independent traditional uses, invalid hybrid relations, and traditional-only fallbacks receive **FAIL**; unresolved findings also receive **FAIL**. Positive structural findings receive **PASS**. An unbound-coexistence notice receives **WARN**, while its independent traditional invocation is separately flagged **FAIL**. Analysis diagnostics receive **WARN**, including application-code diagnostics. Under this policy, a positive finding removes only the associated invocation’s independent-use flag.

Annotation assumptions. We compare the same 50 fixtures with the annotation assumption of Sect. 2.2 enabled and disabled. The disabled setting is the default. Both settings use the same registered API contracts and receiver-type recognition. The fixture specifications use the assumption-enabled setting. Each fixture is scanned with its own directory as the scan root, so its source is treated as application code under the path convention in Appendix A.3.

Environment and reproducibility. The evaluation ran with CPython 3.12.10, PyYAML 6.0.3, Click 8.5.0, and `cryptography` 50.0.1. The reproduction artifact records dependency versions, input hashes, and the JSON/CSV outputs for both modes. Operational reproduction details are in Appendix A.

4.2 Structural Distinctions Beyond Inventory

The two internal heuristics reuse HyBind’s recognized-call records. The flag-all heuristic (A) flags every traditional public-key operation. The project-level PQ-exemption heuristic (B) exempts it whenever the scanned project contains a recognized PQ primitive. Table 5 compares these decisions with HyBind’s structural findings on representative controlled cases. The comparison isolates these decision rules; external analyzers are not evaluated.

For *both component secrets used / PQ secret omitted* and *AND verification / OR verification*, each heuristic retains the same decision within the pair, while HyBind distinguishes their structures. Thus, presence-based exemption alone cannot establish the contribution or acceptance relationship needed to classify a call as a hybrid component. Across the 50 recognized traditional invocations in the assumption-enabled conformance results, A flags all 50, B exempts 43, and HyBind exempts eight. All eight are also exempted by B; HyBind retains flags for the other 35 invocations exempted by B. These are counts on authored fixtures, not estimates of deployment accuracy.

4.3 Conformance and Annotation Sensitivity

With the annotation assumption enabled, all 50 fixtures matched their specified findings, policy exit codes, and application-code diagnostic status.

Diagnostic status and unresolved findings. Table 6 separates application-code diagnostics from unresolved findings. Eleven scans with the annotation assumption and nine default scans contain an UNKNOWN finding without an application-code diagnostic. Provenance or completion evidence can remain unresolved even when no unsupported execution effect is reported.

Disabling the assumption adds diagnostics in three fixtures: an accepting bypass, an independently returned traditional secret and an unresolved outer return. Their conditional parameters lose the primitive-behavior premise, leaving truth conversion unresolved. The other 47 fixtures retain the same findings, policy exits and diagnostic status across settings. For the three affected fixtures, the default setting adds the diagnostic and changes the exit from 1 to 2; the

Table 5. Representative decisions with the annotation assumption enabled. A is flag-all; B is project-level PQ exemption; the HyBind column applies the evaluation policy, which exempts only calls with a positive finding. Rows report decisions about the traditional calls and the corresponding structural evidence.

Controlled case	A	B	HyBind	Finding(s)	Decisive evidence
Standalone RSA encryption/decryption	Flag	Flag	Flag	Traditional use	No hybrid relationship
Unbound coexistence	Flag	Exempt	Flag	Coexistence notice; traditional use	PQ/T calls have no joint result
Both component secrets used	Flag	Exempt	Exempt	Structurally valid	Both feed returned combined key
PQ secret omitted	Flag	Exempt	Flag	Invalid	PQ output absent from combiner
AND verification	Flag	Exempt	Exempt	Structurally valid	Same message origin; joint success
OR verification	Flag	Exempt	Flag	Invalid	Either component can suffice
Traditional fallback	Flag	Exempt	Flag	Invalid fallback; unknown candidate	Separate traditional-only outcome
Different message origins	Flag	Exempt	Flag	Invalid	Distinct parameter origins
Outer accepting bypass	Flag	Exempt	Flag	Invalid	Acceptance outside qualifying pair
Unresolved outer return	Flag	Exempt	Flag	Unknown	Effective decision unresolved
Fixture IDs, in row order: <code>standalone_rsa</code> , <code>coexistence</code> , <code>hybrid_kem_valid</code> , <code>hybrid_kem_missing_secret</code> , <code>hybrid_sig_and</code> , <code>hybrid_sig_or</code> , <code>hybrid_kem_fallback</code> , <code>hybrid_sig_diff_messages</code> , <code>v7_outer_accept_bypass</code> , <code>v7_unknown_outer_return</code> .					

accepting-bypass finding becomes unresolved rather than invalid, and none of the three has a positive finding in either setting. In this suite every scan with an application-code diagnostic contains UNKNOWN, while the converse does not hold.

Table 6. Application-code diagnostics and unresolved findings across the 50 fixtures in each setting. Entries count scans; default disables the annotation assumption.

Application-code diagnostic present?	Contains UNKNOWN?	Assumption enabled	Default
No	No	32	31
No	Yes	11	9
Yes	No	0	0
Yes	Yes	7	10

Within this suite, the annotation assumptions affect diagnostics and unresolved findings, but not the eight fixtures with positive findings. The eight fixtures comprise five key-establishment and three signature-verification cases.

4.4 Boundary Programs and Native API Cases

The second suite contains 33 author-defined programs: 18 boundary programs and 15 native-library programs. Their contracts specify positive-finding counts, including zero where structural evidence is absent or unsupported. This suite measures positive-finding agreement separately from the full-output contract of the conformance suite. The boundary programs cover equivalent direct/Boolean key returns, short-circuited and executed assignments, alternative-secret selection, AND/OR decisions, discarded pairs, multiple verifier pairs, and dynamic truthiness. All 33 match their positive-finding contracts in both settings. The full analysis emits 11 positive findings in 11 programs by default and 13 in 12 programs with the annotation assumption.

The native programs test the same contribution and acceptance criteria on direct library calls. Five receive positive findings in both settings: ML-KEM decapsulation and encapsulation with X25519/HKDF, a key-material alias, and ML-DSA/Ed25519 verification with omitted or explicit `None` context. HyBind withholds positive findings for the remaining ten because of secret omission, swapped encapsulation outputs, rebinding, secret data placed in salt/info, missing verification, accepting failure, different messages, or unsupported context. These outcomes are not all invalid-hybrid findings. The four key programs with omission, rebinding or swapped outputs report unbound coexistence and traditional use; the two salt/info programs report the same with diagnostics. Among the signature programs, missing PQ verification reports traditional use alone, accepting failure and different messages report invalid relationships, and unsupported context produces a diagnostic. All native programs annotate key receivers with directly imported class names (Sect. 5).

Runtime oracles execute the two native positive templates. Verification accepts only the both-valid input among the four component-validity combinations. The receiver’s derived key equals the sender’s independently computed ML-KEM/X25519/HKDF result; dropping either secret or substituting ciphertext changes that result in the recorded execution. A separate dynamic-truthiness oracle rebinds the combiner during `__bool__`, yielding a traditional-only result. The full analysis withholds a positive finding for that program.

4.5 Contribution of the Proof Obligations

Using the same 33 programs and API contracts, we run four experimental variants: replace the universal completion requirement in Eq. (1) by an existential hybrid completion; remove the candidate-specific existential witness requirement in Eq. (2); remove `Eligible(f)`; or remove the latter two together. Table 7 reports finding counts. All variants retain the full analysis’s expected positive findings.

Table 7. Positive finding counts on the 33 boundary/native programs. Added findings include both approvals the full analysis correctly excludes and approvals withheld only by conservative checks; see the text.

Configuration	Default	Assumption enabled
Full analysis	11	13
Existential key completion	11	16
Without candidate witness	11	13
Without function effect eligibility	15	15
Without both effect eligibility and witness	16	16

With the annotation assumption, requiring only one hybrid completion adds findings for three traditional-only-return variants. A hybrid return elsewhere in the function is therefore insufficient to classify its key outcomes. Without the assumption, the effect check independently blocks these cases’ unresolved truth tests.

Removing function effect eligibility admits the dynamic-dispatch program, whose runtime oracle produces a traditional-only result. This case shows why the contribution check depends on stable API bindings during evaluation. Effect removal also admits reachable verifier pairs in two branch programs, one only in default mode. Those additions reflect the coverage cost of the conservative effect check.

Removing the candidate witness alone leaves the counts unchanged because effect eligibility blocks its boundary case. Removing both checks adds a finding for an unreachable verifier pair. A qualifying pair elsewhere in the function cannot supply return evidence for that candidate. On this suite, the witness therefore overlaps with the effect check; these experiments do not isolate its separate contribution.

The artifact preserves all 330 scans, source contracts and runtime-oracle outcomes in `results/revision/`.

5 Limitations and Threats to Validity

Semantic scope. The analysis is intraprocedural and trusts registered API contracts. Native support follows the profiles in Appendix A.7. The native programs annotate key receivers with directly imported class names. Module-qualified receiver annotations evaluated at definition time, and key construction or loading inside the analyzed function, introduce unresolved effects that block positive findings. Calls on receivers without recognized type facts produce no finding, so an empty result does not establish the absence of traditional usage. Registered decryption outputs are outside the key-return obligation (Sect. 3.3). Arbitrary wrappers, cross-file effects and dynamic Python behavior require additional models. Implicit exceptions are not fully enumerated. Candidate acceptance witnesses concern retained return dependencies, not general path satisfiability or soundness for unrestricted Python.

Interpretation of findings. Structural contribution and acceptance do not establish cryptographic binding properties [3], key independence, runtime signature-byte inequality, or end-to-end protocol correctness. The native runtime oracles validate the reported template executions; they are not security proofs of the constructions.

Evaluation validity. The 50 conformance fixtures and 33 boundary/native programs are author-defined. Their specification agreement, runtime checks and component-removal comparisons provide evidence for the implemented distinctions. They do not estimate detection accuracy, repository-scale coverage or UNKNOWN incidence in deployed software.

6 Related Work

Cryptographic misuse analysis. CryptoGuard refines program slices to remove irrelevant dependencies [17]. CCA-Droid combines domain-specific slicing, crypto-state-aware call graphs, and conditional constant propagation [6]. Xia et al. examine LLM-based misuse detection and validation, including errors in cryptographic knowledge and program semantics [19].

Inventories and dependency representations. Cryptoscope connects related API calls using cryptographic knowledge, data/control/call graphs, and contextual slices [7]. Hess describes CycloneDX `dependsOn` edges from a hybrid KEM combiner to its constituents, with discovery and evidence metadata [5]. HyBind’s contribution and acceptance criteria specify the source evidence used to classify an individual traditional invocation as a hybrid component.

Evaluation methodology. MASC systematically varies misuse instances to reveal detector blind spots [2]. Discarded outputs, bypassing branches, and changed message provenance likewise motivate the distinctions in our controlled fixtures.

7 Conclusion

HyBind makes component contribution and signature acceptance explicit criteria for classifying a traditional invocation as part of a PQ/T hybrid. In the controlled evaluations, these criteria distinguish omitted secrets and OR acceptance from joint-secret and AND patterns that the presence-based heuristics treat alike. Relaxing return or effect requirements also admits traditional-only key results, showing why local contribution must be checked against the enclosing function’s outcomes and bindings. The original suite matches all 50 fixture specifications with the annotation assumption enabled, and the separate 33-program suite matches its specified positive-finding counts in both modes. The resulting findings connect individual calls to their contribution and acceptance evidence, while identifying unresolved requirements for migration review. Broader library contracts, interprocedural summaries and an independently labeled repository corpus are needed to evaluate practical coverage beyond the authored suites.

References

1. Alagic, G., Barker, E., Chen, L., Moody, D., Robinson, A., Silberg, H., Waller, N.: Recommendations for key-encapsulation mechanisms. NIST SP 800-227, National Institute of Standards and Technology (Sep 2025). <https://doi.org/10.6028/NIST.SP.800-227>
2. Ami, A.S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., Nadkarni, A.: Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In: 2022 IEEE Symposium on Security and Privacy. pp. 397–414. IEEE Computer Society (2022). <https://doi.org/10.1109/SP46214.2022.9833582>
3. Banerjee, A., Reddy, K. T., Schoiniakakis, D., Hollebeek, T., Ounsworth, M.: Post-quantum cryptography for engineers. RFC 9958, RFC Editor (Jun 2026). <https://doi.org/10.17487/RFC9958>
4. Driscoll, F., Parsons, M., Hale, B.: Terminology for post-quantum traditional hybrid schemes. RFC 9794, RFC Editor (Jun 2025). <https://doi.org/10.17487/RFC9794>
5. Hess, B.: The anatomy of cryptography bills of materials: Standardization and practice in CycloneDX. In: Migration and Agility in Cryptographic Systems. Communications in Computer and Information Science, vol. 3002, pp. 1–19. Springer, Cham (2026). https://doi.org/10.1007/978-3-032-28946-9_1
6. Lee, M., Kim, E., Oh, S., Baek, J., Susilo, W., Kim, H.: CCA-Droid: Context-aware cryptographic API misuse detection in Android apps. In: Proceedings of the ACM Asia Conference on Computer and Communications Security. pp. 1475–1490. ACM (2026). <https://doi.org/10.1145/3779208.3807487>
7. Moffie, M., Boehm, O., Koyfman, A., Bin, E., Sztokman, E., Bhattacharjee, S., Saha, M., McGugan, J.: Cryptoscope: Analyzing cryptographic usages in modern software (2025). <https://doi.org/10.48550/arXiv.2503.19531>, arXiv:2503.19531, version 1

8. Moody, D., Perlner, R., Regenscheid, A., Robinson, A., Cooper, D.: Transition to post-quantum cryptography standards. NIST IR 8547, National Institute of Standards and Technology (Nov 2024). <https://doi.org/10.6028/NIST.IR.8547.ipd>, initial Public Draft
9. National Cybersecurity Center of Excellence: Migration to post-quantum cryptography, <https://www.nccoe.nist.gov/applied-cryptography/migration-to-pqc>, NIST project page. Accessed 7 September 2026
10. National Institute of Standards and Technology: Module-lattice-based digital signature standard. FIPS 204, National Institute of Standards and Technology (Aug 2024). <https://doi.org/10.6028/NIST.FIPS.204>
11. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. FIPS 203, National Institute of Standards and Technology (Aug 2024). <https://doi.org/10.6028/NIST.FIPS.203>
12. Python Cryptographic Authority: Elliptic curve cryptography. cryptography 50.0.1 documentation (2026), <https://cryptography.io/en/50.0.1/hazmat/primitives/asymmetric/ec/>, accessed 27 September 2026
13. Python Cryptographic Authority: ML-DSA signing. cryptography 50.0.1 documentation (2026), <https://cryptography.io/en/50.0.1/hazmat/primitives/asymmetric/mldsa/>, accessed 23 September 2026
14. Python Cryptographic Authority: ML-KEM key encapsulation. cryptography 50.0.1 documentation (2026), <https://cryptography.io/en/50.0.1/hazmat/primitives/asymmetric/mlkem/>, accessed 23 September 2026
15. Python Cryptographic Authority: X25519 key exchange. cryptography 50.0.1 documentation (2026), <https://cryptography.io/en/50.0.1/hazmat/primitives/asymmetric/x25519/>, accessed 27 September 2026
16. Python Software Foundation: The Python Language Reference, Python 3.12, <https://docs.python.org/3.12/reference/>, version-specific online documentation. Accessed 7 September 2026
17. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.D.: CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2455–2472. ACM (2019). <https://doi.org/10.1145/3319535.3345659>
18. van Rossum, G., Lehtosalo, J., Langa, Ł.: PEP 484: Type hints (2014), <https://peps.python.org/pep-0484/>, created 29 September 2014. Accessed 7 September 2026
19. Xia, Y., Xie, Z., Liu, P., Lu, K., Liu, Y., Wang, W., Ji, S.: Beyond static pattern matching? rethinking automatic cryptographic API misuse detection in the era of LLMs. Proceedings of the ACM on Software Engineering **2**(ISSTA), 113–136 (2025). <https://doi.org/10.1145/3728875>

A Supplementary Reproduction and Semantic Scope

The artifact includes the evaluated analyzer, inputs, outputs, and driver. Reproduction uses a temporary copy and verifies input hashes.

A.1 Selected Python semantic boundaries

Table 8 summarizes eligibility rules, conditional on the required operands and bindings being modeled.

Table 8. Selected Python eligibility boundaries.

Construct	Treatment in the current source model
Literal and local name	Tracks literal facts and bindings; a name alone does not establish runtime type.
Annotation	With the assumption enabled, <code>flag: bool</code> supplies a trusted no-user-dispatch premise; <code>object</code> supplies none.
Registered crypto API	Requires recognized binding, arguments, and success/output contracts.
List/tuple literal	Checks element evaluation; insertion does not invoke hashing.
Set element/dictionary key	Literal <code>1</code> passes the exact hash-safe check; a dynamic parameter does not, even with a primitive annotation.
Attribute/subscript	<code>box.message</code> has unresolved provenance; a recognized key receiver alias can still identify a registered method.
Comparison/operator	Supports primitive cases; otherwise dispatch may remain unresolved.
Iteration/unpacking	Literal lists and integer-literal <code>range</code> can be traversed; arbitrary iterators lack this guarantee. The encapsulation tuple uses its output-role contract.
Function/lambda definition	Checks definition-time expressions, including defaults, even if the body is never called.
Exception handling/finally	Checks matching and effective completion; a <code>finally</code> block can replace a pending return or exception.
Imports/helpers	Models selected imports/calls without general interprocedural effect proofs.
Context manager/decorator	Unmodeled entry, exit, or definition-time effects block positive findings; modeled cases such as <code>nullcontext</code> still require eligible arguments.
Dynamic pattern/formatting	Unresolved protocol or formatting effects block positive findings.

A.2 Result fields and denominators

Finding fields follow Sect. 3.5. Evidence levels describe the kind of available evidence rather than confidence scores.

Mode comparisons count each of the 50 fixtures once per predicate. “Contains UNKNOWN” requires at least one unresolved finding; a positive finding requires E3/STRUCTURALLY_VALID for key establishment or verification. These predicates can overlap within a scan. Per-fixture CSV and recognized-call records provide the underlying findings and locations.

A.3 Path conventions and reproduction settings

Path classification is relative to each scan root and uses directory components, not substrings. In precedence order, test markers are `tests`, `test`, and `fixtures`; benchmark markers are `benchmark`, `bench`, and `benchmarks`; example markers are `examples`, `example`, and `docs`; vendored markers are `vendor`, `third_party`, `.venv`, `venv`, and `site-packages`. A filename starting with `test_` or ending with `_test.py` is a test; a filename ending with `_bench.py` is a benchmark. Other files receive the schema label `production`. These rules define the application/auxiliary distinction used by the application-code diagnostic status and operation-level analysis. Each evaluation fixture uses its own directory as the scan root, so its `app.py` is treated as application code even though the suite’s parent is named `fixtures`.

The original suite uses `typed` for the annotation assumption enabled and `default` for it disabled. The additional suite under `results/revision/` names the enabled mode `assumption_enabled`; the underlying option is the same.

The host ran ARM64 macOS (Darwin 25.5.0) with ten logical CPUs. The source policy is `policies/migration.yaml`. It sets `unknown_handling: fail`. Measured dependencies are recorded separately from the development lock. Under this evaluation policy, exit code 0 means that no finding has policy state `FAIL` and no application-code diagnostic is present. Exit code 1 means that at least one finding has policy state `FAIL`, with no application-code diagnostic. An application-code diagnostic produces exit code 2, regardless of finding-level policy states. Exit code 0 does not indicate the absence of unrecognized traditional calls.

A.4 Reproduction entry point

The anonymous archive `HyBind_ICISC2026_artifact.zip` contains the entry point `REVIEW_README.md`. The saved scanner records identify version 0.14.2. From the extracted artifact root, create a Python 3.12.10 virtual environment, install `paper/icisc2026/results/requirements-measured.txt`, and run:

```
.venv-paper/bin/python paper/icisc2026/reproduce/evaluate.py \
--repeats 100 \
--output paper/icisc2026/reproduced-results
```

The `-repeats 100` option sets timing repetitions; timing results are supplementary and described in `SUPPLEMENTARY_COMPONENTS.md`. The archive README explains setup. Input hashes pin the source, fixtures, profiles, and scripts. The output path preserves the supplied results. Table 6 is reproduced by `paper/icisc2026/reproduce/summarize_saved_modes.py` using pinned PyYAML. It reads saved JSON and specification categories, retaining per-fixture predicates, original reasons, and diagnostics as separate axes.

In Table 6, diagnostic present is “No” when `analysis_complete` is true and “Yes” when it is false. Mode labels follow Appendix A.3.

A.5 Annotation and effect-model details

The annotation option marks recognized parameters as having primitive behavior. This marker supplies a no-user-dispatch premise for effect checks, without runtime type tests or a separate type lattice for the six accepted names. Table 9 gives the recognition and invalidation rules.

Table 9. How the annotation option contributes facts to effect checks.

Input form	Treatment
<code>bool</code> , <code>int</code> , <code>float</code> , <code>complex</code> , <code>str</code> , <code>bytes</code>	When enabled, each marks the parameter as primitive; truth conversion, selected operators, and argument-aware built-in checks use this marker.
Qualified name ending in one of these six names	Recognized by its final name component; this recognition does not resolve or validate the annotation object’s runtime identity. This syntactic marker rule is separate from exact-symbol API-contract recognition.
Missing, quoted-string, container, or other annotation	Supplies no primitive marker through this rule. Literal evaluation and registered API/receiver facts remain separate sources of information.
Rebinding or deleting a marked parameter	Removes its annotation-derived marker for the function, rather than assuming the original fact remains true after the write.

Parameter scanning includes positional, keyword-only and variadic parameters. For `*args` and `**kwargs`, the marker applies to the local tuple or dictionary name, without separating container and element facts. The trusted premise must therefore cover element-level dispatch in the marked operations; the implementation does not verify this recursively. Subscription remains unresolved, and set elements or dictionary keys still require hashable literals.

Receiver annotations such as `ec.EllipticCurvePublicKey` are resolved separately against known library types in both modes. That recognition identifies a contract; it is not a check of caller behavior or runtime effect conformance.

Built-in names alone do not imply effect-free calls. For example, `len(b"x")` has literal arguments, whereas `len(x)` with an unmarked parameter can dispatch user code. The argument-aware model covers `len`, `bool`, `str`, `bytes`, `int`, `float`, `list`, `tuple`, `set`, `frozenset`, `sorted`, `reversed`, `enumerate`, `zip`, `all`, `any`, `max`, `min`, and `abs`, with separate rules for `range`, `isinstance`, and selected exception constructors. Argument evaluation is checked too. A hash-safe set element must additionally be a hashable literal, such as `1` or a tuple of literal values; an annotated name alone does not satisfy that stronger test.

Dispatch and exceptions. Primitive markers resolve dispatch effects, but implicit exception coverage remains limited. Binary-operator checks do not test zero divisors or other built-in error conditions. Key-value transfer visits operands without creating operator-specific exceptional completions. The signature evaluator computes selected literal operations and maps caught evaluation errors or unsupported operators to unknown values. Try summaries traverse handlers from the incoming state rather than matching individual arithmetic exceptions to handlers.

A.6 Completion and explanation-record details

The included key-return summaries are `hybrid_key`, `traditional_key`, `pq_key`, `unknown_key`, and `unsupported`. These are outcome categories, not runtime key types. For resolved, supported summaries, both producer families and retained hybrid status yield `hybrid_key`; a single family yields its corresponding category. A return whose value summary has the explicit `non_key` kind is classified as `other` and excluded, as is an escaping `raise`. The non-key kind covers registered encryption/decryption, signing and verification results, and guaranteed Boolean results of `not`, identity and membership checks. Boolean selection preserves operand dependencies and literal short-circuit effects. Exact bytes addition retains contributing producers when its local contract-derived type facts hold; literal byte padding adds no secret contribution. Other binary/unary operations and rich comparisons preserve dependencies as unresolved values. Unclassified constants, containers, and missing values remain unresolved.

Finally transfer and eligibility. The completion transfer runs the `finally` block on continuing states and on pending returns or exceptions. Fall-through preserves an already evaluated pending return; a new completion replaces it. Separately, the scope check can withhold positive findings when a potentially overriding block follows recognized cryptographic calls in the try/else body, or reads a try-assigned value not definitely bound on every incoming path. Key-completion replacement can also add a diagnostic.

The final record check is a record consistency check for provisional positive key and signature findings. The field `scope_semantics_eligible` must be true. The check requires the effect flags in Table 10 to evaluate true and the two unresolved lists to be empty. These fields summarize the preceding provenance, completion and effect analyses.

Table 10. Fields consumed by the common record check and their defaults.

Field	Value used if absent
<code>execution_effects_complete</code>	Scope eligibility
<code>api_binding_state_complete</code>	Scope eligibility
<code>runtime_effects_complete</code>	Execution-effect flag
<code>implicit_protocol_effects_complete</code>	Runtime-effect flag
<code>builtin_argument_safety_complete</code>	Runtime-effect flag
<code>namespace_alias_state_complete</code>	Binding-state flag
<code>unsupported_effects</code>	Empty list
<code>unresolved_required_fields</code>	Empty list

Finding constructors populate these fields; omitted flags inherit the listed defaults. A failed check changes the finding to E2/UNKNOWN, clears its component association, and applies the policy for unresolved results.

Signature records include unresolved requirements and, for positive findings, candidate witness locations and verifier-call IDs. Key records include each outcome’s location, category, supported-state flag and producer IDs. Every included key outcome must be hybrid. Both feed the common consistency check.

A.7 Registered API profiles

The registry contains 36 exact-symbol contracts: 19 fixture interfaces, 10 traditional methods from `cryptography`, four ML-KEM-768/ML-DSA-65 methods and three SHA256/HKDF constructor or derivation contracts. The fixture encapsulation wrapper returns ciphertext followed by shared secret; native ML-KEM returns shared secret followed by ciphertext [14]. The collector uses these output roles when resolving tuple assignments. Native ML-DSA verification raises `InvalidSignature` on failure; the supported context is omitted or literal `None` [13].

HKDF requires a locally validated constructor or tracked alias with SHA-256, 32-byte output, literal bytes/`None` salt and literal bytes info. Receiver annotations identify the class contract but do not establish these settings. Only the `derive` key-material argument contributes secrets. Call recognition uses bindings at the call site; rebinding invalidates facts, and branch joins retain facts shared by all continuing paths.

The bytes rule tracks producer-derived operands through annotated assignments, local assembly and returns. Literals add no secrets. Single-family intermediate steps used only by further bytes assembly are excluded as candidates; unresolved derived inputs receive UNKNOWN. General combiner/KDF outputs and branch alternatives remain opaque to this rule.