

Hyde: Search-Based Hypertesting of Observational Determinism in Java Programs

Alexander Chevalier

▶ **To cite this version:**

Alexander Chevalier. Hyde: Search-Based Hypertesting of Observational Determinism in Java Programs. 2026. ⟨hal-05703432⟩

HAL Id: hal-05703432

https://hal.science/view/index/docid/5703432

Preprint submitted on 24 Jul 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY-NC-ND 4.0 - Attribution - Non-commercial use - No Derivative Works - International License

HYDE: Search-Based Hypertesting of Observational Determinism in Java Programs

ALEXANDER CHEVALIER, Department of Computer Science, Paris, France

Software testing is overwhelmingly concerned with *trace properties*: requirements whose satisfaction can be decided by examining one execution in isolation. An important class of security and concurrency requirements lies beyond this frame. *Non-interference* and *observational determinism* are *hyperproperties*. They constrain the relationship between several executions, and so cannot be refuted by any single run. This paper presents HYDE, an automated tool that dynamically searches for violations of observational determinism in concurrent Java. Instantiating and extending the hypertesting framework for a non-deterministic setting, HYDE reformulates refutation as an optimisation problem over *pairs* of low-equivalent inputs. A many-objective search evolves such pairs under a two-part fitness. The fitness first drives the two runs to jointly reach the program locations where a divergence can surface, namely writes to public state and thread joins, and then maximises the divergence between their public histories, in the spirit of differential side-channel fuzzing. To attack a scheduler-independent property with quantified confidence, the search samples interleavings through a randomised (PCT-style) scheduler rather than trusting the native one. A source-level instrumenter, realised as an OpenRewrite recipe, records both the coverage heuristics and the time-ordered public histories that an oracle compares up to stuttering and prefixing, returning a minimal replayable witness. Because a verdict is issued only after a witnessing pair has been executed, HYDE is free of false positives by construction. We evaluate HYDE through an automated protocol on a labelled suite of concurrent subjects, and position it against the public IFSpec information-flow benchmark and taint-analysis baselines. The sole residual limitation is one of scope. Purely time-based channels, in which the secret affects timing but no public value, are not observed by a value-based oracle.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Program analysis*.

Additional Key Words and Phrases: hyperproperties, observational determinism, non-interference, search-based testing, hypertesting, concurrency, program instrumentation

1 Introduction

Contemporary software is expected to satisfy requirements that reach well beyond input–output correctness: it must protect confidential data [21, 43], avoid data races [26, 53], and behave predictably irrespective of how a scheduler interleaves its threads [10, 36]. Software Verification & Validation, and testing in particular, is well suited to the first kind of guarantee. The requirement is expressed as a *trace property* [18], that is, as a set of admissible executions. A test either exhibits an offending run or it does not. A substantial category of requirements, however, cannot be captured in this manner, because their very statement quantifies over *several* executions simultaneously.

Confidentiality is the canonical illustration. To establish that a program never allows a secret to reach a publicly observable channel, one execution does not suffice: the public inputs must be fixed, the secret varied, and the resulting public outputs compared [22, 28, 51]. A discrepancy constitutes a proof of leakage. This requirement is *non-interference* [20, 38], and it is relational by nature, relating two executions. Concurrency compounds the difficulty. Since a scheduler may interleave threads differently on identical inputs, no property that quantifies over "all interleavings" can be decided from a single observed run. *Observational determinism* (OD) [50, 62] is the canonical concurrent instance. It requires that the sequence of public values a program emits be independent of both the secret and the schedule, and it generalises non-interference to multi-threaded code.

Author's Contact Information: Alexander Chevalier, Department of Computer Science, Paris, France.

Requirements of this form are *hyperproperties* [18]: whereas a trace property is a set of traces, a hyperproperty is a set of *sets* of traces, and can thus constrain how executions relate to one another. This formalisation gave rise to dedicated logics [17] and to a substantial body of static verifiers based on model checking [24, 32] and abstract interpretation [3]. Static techniques, however, inherit well-known limitations: only restricted, decidable fragments are tractable [24, 56], and sound over-approximation tends to yield a considerable number of false positives that impede adoption in practice [8, 34]. Dynamic techniques, which reason about concrete executions, forgo completeness in exchange for precision and scalability; yet they remain comparatively under-explored for hyperproperties, and more so once genuine concurrency is involved [35].

This paper contributes Hyde (HYperproperty Dynamic tEsting), a dynamic, fully automated tool that pursues OD by *constructing* counterexamples. Hyde treats the search for a violation as an optimisation problem over *pairs* of low-equivalent inputs and, in doing so, embraces the discipline of *hypertesting* [35], extending it to the non-deterministic, concurrent regime that OD demands. Concretely, the paper:

- surveys the hyperproperty landscape and the foundations of hypertesting (Section 2), and situates Hyde among static and dynamic approaches (Section 3);
- formulates OD refutation as a search problem and introduces *witness points*, the locations whose joint coverage by a pair of runs can expose an OD violation, together with a paired objective and genetic operators that preserve low-equivalence to guide the search (Section 4);
- describes a compact prototype whose source-level OpenRewrite instrumenter serves both the coverage heuristics and a history-comparing OD oracle (Section 5); and
- evaluates Hyde on concurrent subjects, positions it against the public IFSpec benchmark and taint-analysis baselines, and documents a structural limitation (Section 6).

2 Background

This section establishes the conceptual foundations on which Hyde builds: the theory of hyperproperties and its logics, information-flow security and non-interference, observational determinism, and the notions of hypertesting and search-based test generation.

2.1 Trace properties and hyperproperties

A trace property is a set of executions; a program satisfies it when each of its runs is a member of the set, so satisfaction is decided run by run. Clarkson and Schneider [18] observed that a broad range of correctness and security requirements are not of this form, and introduced *hyperproperties*: sets of sets of traces, capable of predicating over the entire semantics of a program rather than over individual runs. Just as trace properties decompose into safety and liveness, hyperproperties decompose into *hypersafety* and *hyperliveness* [18, 19]. Of particular practical significance is *k-safety*: a hyperproperty refutable by exhibiting k mutually inconsistent traces [57]. When k is small the notion is amenable to testing, since a finite tuple of concrete runs constitutes a complete counterexample; non-interference and OD are both 2-safety.

The specification and verification of hyperproperties motivated dedicated *hyperlogics*. HyperLTL and HyperCTL* [17] augment linear- and branching-time temporal logic with explicit quantification over traces, and model-checking algorithms exist for them, both exact [24] and bounded [32]; relational program logics discharge k-safety obligations deductively [6, 56], and runtime monitors observe streams of executions to detect violations online [1, 11, 12, 25]. The scope of hyperproperties extends considerably beyond confidentiality: they express robustness and planning objectives for robotic and multi-agent systems [9, 61], quantitative guarantees for

hardware [52], the preservation targets of secure compilation [48], and, by regarding the original and mutated program as two runs, mutation testing itself [23]. This breadth is precisely what makes dynamic, scalable checkers attractive, since the static machinery, however elegant, decides only restricted fragments and tends toward conservative verdicts.

2.2 Information-flow security and non-interference

Language-based confidentiality partitions program data according to a lattice of security levels [21, 51]; for present purposes the two-point lattice $\mathit{low} \sqsubseteq \mathit{high}$, separating public from confidential data, is sufficient. Non-interference originates with Goguen and Meseguer [28] and with early work of Denning [20, 22]. It requires that public outputs depend on public inputs alone. Writing $\sim_L$ for the agreement of two states on their public component, a terminating program P is non-interferent when

$$s_1 \sim_L s_2 \implies P(s_1) \sim_L P(s_2).$$

Because two runs are involved, non-interference is a canonical 2-safety hyperproperty [57]. It has traditionally been enforced by security-type systems [22, 43, 60] and characterised over execution traces [38]; weaker, quantitative variants relax the all-or-nothing view and bound *how much* a secret may leak, using entropy or channel capacity [16, 55]. A recurrent engineering expedient approximates this relational property by a trace property: *taint analysis* [7, 54] propagates labels from secret sources to public sinks along a single run. The approximation is convenient but imprecise. A label may reach a sink without genuinely influencing it (over-tainting), producing false alarms that a properly relational check would avoid. Self-composition [5, 57] instead preserves the relational meaning by verifying P against a copy of itself, reducing the 2-safety obligation to an ordinary safety one; dynamic monitors enforce related policies at run time [4, 37].

2.3 Observational determinism

Non-interference presupposes determinism and does not transfer directly to concurrency: two runs on identical inputs may already differ solely because the scheduler selected a different interleaving, without any secret being implicated. Observational determinism [62] addresses this by requiring the *observable behaviour* to be independent of both the secret and the schedule, where the observable behaviour is the sequence of values that public variables assume over time. It is a scheduler-independent security notion: it must hold even against an adversary that controls thread scheduling.

Formally, an execution is modelled as the sequence of memory states $\tau = M_0 M_1 \cdots$ it traverses, with $\tau(\ell) = M_0(\ell) M_1(\ell) \cdots$ denoting its projection onto a location ℓ. Two traces are low-equivalent, $\tau_1 \approx_L \tau_2$, when for every public location the projections agree modulo two relaxations that jointly define the attacker model:

Stuttering. Maximal runs of an unchanged value are collapsed to a single occurrence, rendering the observer insensitive to timing and to silent intermediate steps [33, 58].

Prefixing. Two histories are compared up to the length of the shorter one, permitting a non-terminating branch to leak at most one bit and dispensing with a termination proof; a well-known consequence is that $\approx_L$ is not transitive.

The adversary is therefore a passive observer that may replay the program at will and read public variables at any instant, which is strictly stronger than the input/output observer of ordinary non-interference. OD holds when every pair of low-equivalent inputs yields low-equivalent traces. Being 2-safety, it fails as soon as a single pair does not. As Zdancewic and Myers observed, a bare race on a public variable suffices to violate OD even in the absence of any secret: in $\ell \leftarrow 0 \parallel \ell \leftarrow 1$ the final public value is scheduler-dependent. Static verification of OD has been pursued with

confluence-based type systems [62], less restrictive type systems [58], and temporal-logic characterisations [33], all subject to the customary trade-offs of static analysis.

2.4 Hypertesting and search-based generation

Kinder [35] argued that hyperproperties warrant a testing discipline of their own, which he called *hypertesting*. This section fixes the notions that such a discipline requires and that the rest of the paper uses.

A *relational k-bounded hyperproperty* (k-hyperproperty) takes the form

$$\forall e_1 \cdots \forall e_k.\ P_{\text{in}}(e_1^{\,i}, \ldots, e_k^{\,i}) \Rightarrow P_{\text{out}}(e_1^{\,o}, \ldots, e_k^{\,o}),$$

where $e^{\,i}$ and $e^{\,o}$ denote the input and output memories of an execution and $P_{\text{in}}, P_{\text{out}}$ are k-ary predicates. Non-interference is the instance k=2 in which both predicates are low-equivalence. Refuting a k-hyperproperty requires k appropriately chosen runs; accordingly, a *hypertest input* is a tuple of k inputs satisfying P_{in}, a *hyperoracle* decides P_{out} on the resulting outputs, and adequacy is measured by *hypercoverage*. A hypercoverage goal is a tuple of k program points, each a definition of an output-relevant variable or a point transitively data-dependent on one, that the k runs must cover *jointly*. Covering a goal is necessary, though not sufficient, for a violation; goals that no admissible tuple can cover (*infeasible*) or whose points always assign equal output values (*irrelevant*) are redundant, and may be pruned, approximately, by forward slicing from the inputs and by constant propagation. To direct generation toward uncovered goals, the classical single-run heuristics of search-based testing [27, 30, 39], namely approach level and branch distance [2, 59], are lifted from one run to k, and generation is cast as a many-objective optimisation adapting the MOSA algorithm [46]. Two assumptions are implicit in this reading, and they are exactly the ones OD breaks: that subjects behave *deterministically*, and that observing inputs and outputs suffices. OD is defined over *intermediate* public state under a *non-deterministic* scheduler. Reconciling the two is the subject of the next section.

3 Related Work

Static verification. The majority of hyperproperty verification is static. Hyperlogics [17] are model-checked exactly [24] or in bounded form [32]; relational program logics discharge k-safety obligations deductively [6, 56]; and abstract interpretation contributes a hypercollecting semantics for information flow [3]. For OD specifically, verification has relied on confluence-based and refined type systems [58, 62] and on temporal-logic characterisations [33]. All confront restricted decidable fragments and conservative verdicts on realistic concurrent code. These are the pressures that a dynamic complement is intended to relieve.

Dynamic and runtime approaches. Runtime monitors for HyperLTL observe streams of executions and flag k-safety violations online [1, 11, 12, 25]; unlike Hyde, they consume executions supplied to them rather than generating adversarial ones. A fuzzing line targets information-flow and side-channel leaks. Examples include DifFuzz [44], QFuzz [45], ct-fuzz [31], JIT side-channel fuzzing [13], and SoC hyperfuzzing [42]. These tools proceed predominantly through unguided differential or self-composed execution, without a coverage notion tailored to the relational goal. Genetic-improvement tools such as HyperGI [40] and its successor [41] employ entropy estimates to detect and *repair* leakage, emphasising patching over counterexample generation. Dynamic confidentiality monitors [4, 37] and dynamic taint analysis [7] approximate non-interference by single-run tracking and, as discussed, are prone to over-reporting. None of these targets OD under a non-deterministic scheduler.

Concurrency testing and race detection. It is tempting to reach for a data-race detector such as Eraser [53] or FastTrack [26], but a race is neither necessary nor sufficient for an OD violation, so such a tool solves a different problem. It is not sufficient, because a race on a public cell need not break OD: if the racing writes store the same value, as in the secure baseline of Section 6, the observable history is identical under every schedule even though a detector flags the race. It is not necessary, because OD can fail with no race at all: the sequential leak of Listing 2 is single-threaded, and a fully synchronised program whose public output still depends on the secret leaks without any racing access. A race detector reasons about the memory model, whereas Hyde reasons about the observable value history, which is a 2-safety hyperproperty. Hyde does borrow the schedule-exploration machinery of concurrency testing, sampling interleavings with a PCT-style randomised scheduler [14] to expose the schedules under which a genuine OD counterexample manifests.

Differential fuzzing, and what Hyde adds. DifFuzz [44] and QFuzz [45] are the closest dynamic techniques, and Hyde's divergence term is a direct descendant of their difference-maximising objective. Three things separate Hyde from them. First, they observe a scalar *resource cost*, such as time or space, over a self-composed pair, whereas Hyde's oracle observes the *value history* of the public state and decides OD up to stuttering and prefixing. Second, they maximise a difference with no coverage notion for the relational goal, whereas Hyde guides the search with witness points and the hypercoverage view, so the two runs are first driven to the locations where a divergence can appear and only then pushed apart. Third, they assume deterministic, sequential cost behaviour, whereas Hyde searches interleavings under a randomised scheduler and observes intermediate concurrent state. The two views are complementary: folding a resource cost into Hyde's divergence term would extend it to the timing channels that lie outside a value-based oracle (Section 6).

Hypertesting. A systematic treatment of hypertesting for *deterministic* programs is due to Pasqua et al. [47], evaluated on IFSpec [29] against taint analysis, and from it we take the notion of hypercoverage and the idea of a paired structural heuristic. That setting assumes determinism and observes program endpoints only. Hyde targets a concurrent, scheduler-dependent hyperproperty instead, and contributes the OD-specific witness points, the divergence term, the PCT-scheduled search, and the history-comparing oracle of Section 4. More broadly, hyperproperties subsume metamorphic [15] and mutation [23] testing, secure compilation [48], and quantitative guarantees [52], which suggests that the witness-point-plus-paired-objective recipe could be retargeted well beyond OD.

4 Design

Hyde decides OD by attempting to construct a counterexample: a pair of low-equivalent inputs whose executions leave behind distinguishable public histories. Figure 1 depicts the pipeline. This section develops its conceptual core: the locations the search targets, the state it observes and the rationale for doing so, the objective that guides it, and the evolutionary procedure and oracle that close the loop. The conceptual weight is deliberately concentrated here; Section 5 contributes only the concrete engineering.

4.1 OD as a refutation problem

OD is a 2-safety hyperproperty, and hence a single pair of observationally distinguishable executions refutes it. Cast in the k-hyperproperty schema of Section 2 with k=2, the input predicate is low-equivalence of the two inputs and the output predicate is low-equivalence of the two *traces*,

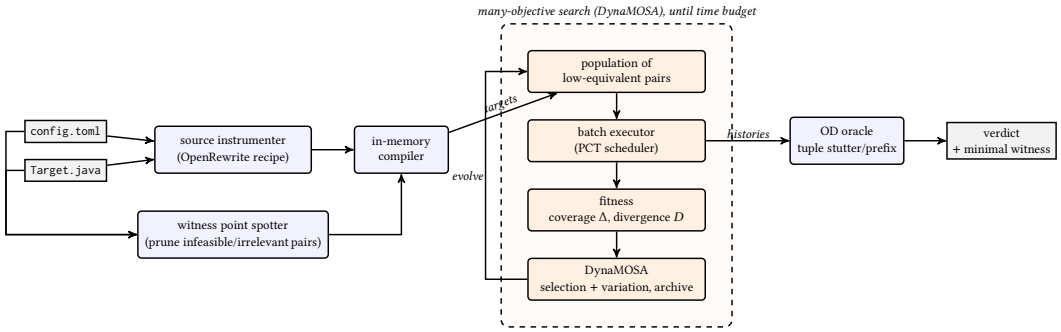


Fig. 1. The Hyde pipeline. From a target method and its high/low annotation, the OpenRewrite recipe weaves the coverage and history probes while the spotter enumerates witness point pairs and discards the infeasible and irrelevant ones. The many-objective loop (DynaMOSA) evolves pairs of low-equivalent inputs toward those targets, executing each under a randomised PCT scheduler and scoring it by a two-part fitness: first the coverage distance Δ to the witness points, then the divergence D between the public histories. The oracle compares the recorded tuple histories up to stuttering and prefixing, returning a verdict and, on a divergence, a minimal replayable witness.

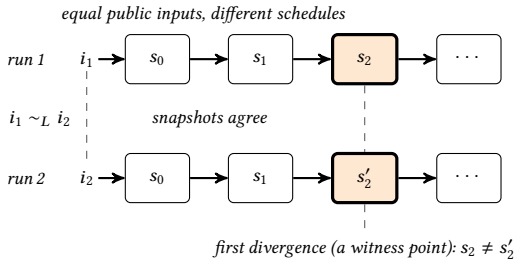


Fig. 2. The general shape of an OD counterexample. Two low-equivalent inputs $i_1 \sim_L i_2$ are run under two schedules. Each run yields a sequence of public state snapshots $s_0 s_1 s_2 \cdots$, that is, tuples over the public locations, stutter-collapsed. The runs agree on a prefix and then disagree at a witness point, where $s_2 \neq s'_2$. That earliest disagreement is the witness the oracle reports.

rather than merely of the two outputs. This trace-level output predicate is the essential departure from the non-interference instantiation: an OD counterexample is a pair of low-equivalent inputs whose *histories* of public values diverge under stuttering and prefixing. The testing campaign therefore seeks such a pair, and its adequacy is assessed by the extent to which pairs of runs jointly cover the locations at which such a divergence can become observable.

4.2 Witness points and goal pruning

For a 2-safety hyperproperty the meaningful search targets are pairs of locations that a pair of runs must reach together for a divergence to surface. We term such a location a witness point. For OD, witness points are of two kinds.

Definition 4.1 (Witness points for OD). A program location is a witness point for observational determinism if it is either (i) a *write to a public variable*, where a secret becomes observable should it flow, or (ii) the *final join of the worker threads*, after which a scheduling race, if present, has already determined the public outcome.

Clause (i) is the concurrent counterpart of non-interference; clause (ii) is the contribution of concurrency, since threads that write a shared public cell in an unconstrained order may leave different values behind even in the absence of any secret, and the appropriate moment to observe this is immediately after the last join. Because OD is 2-safety, the campaign aims to have a pair of runs cover a *pair* of witness points, and a divergence in the histories recorded there refutes the property.

Not every pair of witness points is worth pursuing. Following the hypercoverage view of Section 2, a pair is *infeasible* when no admissible input pair can reach both locations, and *irrelevant* when the public writes there invariably produce equal values, so that they can never disagree. HYDE exploits the inexpensive end of this spectrum directly: it discards pairs whose written value is a compile-time constant (a lightweight constant-propagation check), since such locations cannot contribute a divergence. Feasibility pruning by forward slicing from the inputs is admissible for the same reason, since a redundant pair can never witness a violation, and it integrates cleanly. It is not required for the subjects considered here. When a method exposes fewer than two witness points, the tool degrades gracefully, pairing a solitary witness point with itself or, in their absence, forgoing the gradient in favour of unguided random pairs that the oracle still adjudicates.

A method with m witness points induces $\binom{m}{2}$ candidate pairs, so a naive enumeration is quadratic and does not scale to large methods. HYDE never materialises the full set. Pruning removes the constant-valued and unreachable witness points first, which shrinks m before any pairing. The many-objective search of Section 4.6 then selects targets *dynamically*: it works only on the frontier of currently reachable, still-uncovered pairs, in the manner of DynaMOSA's dynamic target selection [46], and retires a pair to the archive once covered. When m is nonetheless large, the frontier is capped and the remaining pairs are sampled rather than enumerated, biased toward witness points that past runs found easy to reach. The quadratic set is thus a conceptual upper bound, not a quantity the tool ever holds in memory.

4.3 What to observe, and why

Two quite different quantities must be extracted from each run: the structural information required by the search, and the public history consumed by the oracle. HYDE collects them through two independent families of probes.

The first supplies the search heuristics. At every decision the run reports its *branch distance*, a measure of how close the guard came to steering control toward the target. It also reports sufficient control-flow information to compute the *approach level*, the number of nested control dependences of the target left unsatisfied. These are the standard structural meta-heuristics [39, 59], here evaluated against witness points (Figure 3).

The second reconstructs the observable behaviour that OD concerns. Conceptually it is a map from each public location to the ordered list of values it assumes over time. A value is recorded at every write to a public location and, once more, after the *last* thread join; identifying the last join reduces to tracking the live threads and emitting a milestone once none remain. Writes that occur inside a thread are recorded as well, so that the history reflects the intermediate public state and not only the endpoint. To keep that log meaningful under concurrency, each public write and its recording pass together through a single monitor, which makes the recorded sequence linearisable. The two OD relaxations are applied at comparison time rather than at recording time: consecutive duplicates are collapsed (stuttering) and the two histories are compared up to the shorter length (prefixing). One cost of recording in-thread is a probe effect, which we quantify against and mitigate in Section 4.6.

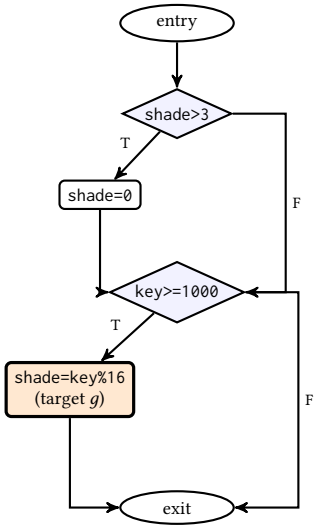


Fig. 3. Control-flow graph of the `tag` subject (Listing 2). The target public write is control dependent on `key>=1000`: for a run that stalls at that predicate the approach level is 0 and the branch distance measures how far key lies from 1000, furnishing the search with a gradient toward the witness point.

4.4 From variables to objects

The development so far speaks of "public variables", but nothing in the method is tied to a particular kind of variable. Witness points are writes to public *locations* and thread joins; the oracle compares the public history over those locations; the fitness rewards divergence there. All of this is stated over locations, so the same machinery applies unchanged to the fields of objects and to cells on the heap. What changes is only how a location is labelled public or secret.

For primitive and atomic parameters the labelling is immediate, and this is the annotation surface the prototype exposes. For object fields it must contend with aliasing. Two references may denote the same object, so a field reached through a "low" reference and through a "high" reference is one and the same cell, and labelling it consistently requires knowing that the two references alias. The method resolves this by reading the annotation through a points-to or alias analysis: each field access is mapped to the set of abstract heap locations it may touch, and a location is deemed public only when every access that may reach it is low. With that resolution in place, an object field is just another public location, its writes are witness points like any other, and the coverage-and-divergence search and the oracle operate without modification. The step from variables to objects is therefore a matter of resolving the annotation, not of changing the method.

4.5 A coverage-and-divergence fitness

An individual is a candidate counterexample: an ordered pair of inputs (i_1, i_2) with $i_1 \sim_L i_2$, since only low-equivalent pairs can falsify OD. Its fitness has two components, mirroring the two things a counterexample must accomplish. It must *reach* a pair of witness points and, there, *diverge*.

Coverage. The first component measures how close the pair comes to jointly covering a target pair of witness points (g_1, g_2). Over the induced traces (τ_1, τ_2) we use the single-run distance

$$\delta(g, \tau) = al(g, \tau) + \widehat{bd}(g, \tau),$$

with *al* the approach level and $\widehat{bd}$ the branch distance normalised into $[0,1)$ by $x \mapsto x/(x{+}1)$ [2], and combine the two runs by the cheaper assignment of traces to targets,

$$\Delta\big((g_1,g_2),(\tau_1,\tau_2)\big) = \min\Big\{\delta(g_1,\tau_1)+\delta(g_2,\tau_2),\Big\}, \\ \delta(g_1,\tau_2)+\delta(g_2,\tau_1)$$

so that which run of the pair covers which target is immaterial. Driving $\Delta \to 0$ brings both witness points within joint reach.

Divergence. Coverage is necessary but not sufficient: a pair may reach both witness points yet write identical values, exhibiting no divergence and offering the search no further gradient. We therefore add a second component that, once the targets are reached, rewards pairs whose public histories *disagree as widely as possible*. Concretely, for each public variable v let $\sigma_i(v)$ be the stutter-collapsed sequence of values v takes in run i, and let

$$d_v \;=\; 1 - \frac{|\mathrm{LCS}(\sigma_1(v),\sigma_2(v))|}{\max(|\sigma_1(v)|,|\sigma_2(v)|)} \;\in [0,1]$$

be one minus the length-normalised longest-common-subsequence similarity of the two sequences. We aggregate over the public variables V by the maximum,

$$\mathcal{D}(\tau_1,\tau_2) \;=\; \max_{v\in V} d_v \;\in [0,1].$$

The maximum is the natural choice for a refutation: OD fails as soon as *some* public variable diverges, so taking the maximum keeps the strongest single divergence undiluted, whereas a sum or a mean would let many equal variables mask one divergent one. (Sum and mean remain admissible alternatives when a broader, less peaked gradient is preferred.) By construction $\mathcal{D} = 0$ precisely when every public variable is low-equivalent, and $\mathcal{D}$ increases with the extent of the divergence. This is the relational analogue of the difference-maximising objectives that make differential side-channel fuzzers effective: DifFuzz steers inputs toward a maximal cost difference $\delta = |c(P_1) - c(P_2)|$ between secret-dependent runs [44], and QFuzz maximises the number of distinguishable observations [45]. Hyde transposes the idea from a scalar resource cost to the observable public history over which OD is defined.

Combined objective. Hyde realises the two components as a *many-objective* problem, since the search of the next subsection is many-objective. Each individual is scored by the vector

$$\big(\,\delta(g_1),\;\delta(g_2),\;1-\mathcal{D}\,\big),$$

whose first two entries pull the two runs onto the witness points and whose third rewards divergence once they are reached. The optimiser keeps the pairs that are non-dominated along the reachability and divergence axes, rather than collapsing them into a single number prematurely. When a single scalar is instead wanted, for a classical genetic algorithm or for reporting, the same intent is captured by minimising

$$\mathcal{F} \;=\; \Delta \;-\; \mathbf{1}[\Delta = 0]\cdot\mathcal{D},$$

which optimises coverage first and divergence second. We use the vector form throughout, and regard $\mathcal{F}$ only as its single-objective reformulation.

4.6 Many-objective search with schedule perturbation

Two choices distinguish Hyde's search from a plain genetic algorithm.

First, instead of optimising a single scalar, Hyde treats each witness-point pair, together with its divergence, as a separate objective and searches with a many-objective algorithm in the DynaMOSA family [46]: it selects targets dynamically, ranks individuals by preference and non-dominated sorting, preserves diversity through crowding distance, and maintains an *archive* of the best pair discovered for each objective. This scales to the many witness-point pairs a method may expose and resists the premature convergence to which a single aggregate objective is prone. The genetic operators respect low-equivalence by construction: *crossover* exchanges material only in the secret dimension, so that offspring remain low-equivalent, and *mutation* perturbs a secret in one run in isolation but mirrors a perturbed public value onto both runs.

Second, and specific to concurrency, a race surfaces only under particular interleavings, so searching over inputs alone does not suffice: the schedule must be searched as well. Hyde therefore does not defer to the native scheduler but drives each execution through a randomised scheduler in the style of PCT [14], which inserts a small number of priority-change points at the instrumented locations and thereby furnishes a *quantified* lower bound on the probability of exhibiting a bug of a given depth. Each individual is evaluated over several sampled schedules with distinct seeds, and its divergence score aggregates the observations across them; a pair is declared a counterexample as soon as any sampled schedule makes the histories disagree. Coupling the input search with schedule sampling is what allows a dynamic tool to attack a scheduler-independent property with quantified confidence rather than by chance.

4.7 Oracle

For every executed pair the oracle decides the low-equivalence of the two public-value histories. Rather than examine each variable in isolation, it compares the *tuple* of public locations up to stutter-equivalence, collapsing maximal runs of identical tuples and matching the two sequences up to the length of the shorter one (prefixing). This choice deserves a word, because Zdancewic and Myers [62] originally phrased OD *per variable*: each public location's sequence must be stutter-equivalent across the two runs, aligned independently of the others. The per-variable reading models an attacker who observes each variable on a separate channel with its own clock. Our attacker, fixed in Section 2, is stronger: it reads the whole public state at each instant, and so perceives the joint sequence of tuples. Comparing tuples is the faithful oracle for that attacker. It is also the more conservative one, since tuple stutter-equivalence implies the per-variable version but not conversely, so a program may satisfy the per-variable definition yet leak through the *correlation* between two public variables. That gap was among the reasons the original definition was later refined [33]. The tuple oracle remains free of false positives: a divergence between the two tuple sequences is a genuine difference in what the joint attacker observes, not an artefact of the alignment. A violation is reported exactly when the two stutter-collapsed sequences disagree, i.e. when $\tau_1 \not\approx_L \tau_2$; the oracle returns the earliest position at which they differ, together with the offending variable and the two values, so that every UNSAFE verdict is accompanied by a minimal, replayable witness. A disagreement admits one of two causes. Either a secret has reached a public write, or a race has reordered public updates. Either constitutes an OD violation.

The histories the oracle compares include the writes made *inside* threads, so an intermediate divergence between two runs is visible and not only a difference at the endpoint. Section 4 explained how the log stays sound under concurrency: each public write and its recording pass through one monitor, making the sequence linearisable. The one caveat is a *probe effect*, since serialising writes through the monitor perturbs timing and can suppress interleavings that the unsynchronised program would exhibit. Hyde counters this by pairing in-thread logging with the PCT scheduler above, which re-introduces interleaving diversity by design rather than by accident, and by enabling in-thread logging only for the public locations named in the descriptor. Purely time-based channels,

in which the secret affects how long a computation runs but never a public value, lie outside the remit of a value-based oracle by definition and are the province of resource-guided fuzzers [44, 45]; we return to this in Section 6.

Overall, the procedure returns UNSAFE with a witnessing pair when the oracle fires; otherwise, if a high proportion of witness-point pairs has been covered across the sampled schedules without any divergence, it reports LIKELY-SAFE; and if coverage remains low, it GIVES UP. Being a concrete-execution method, Hyde never raises a false alarm: a reported divergence is a genuine counterexample. As a dynamic technique it remains incomplete, though the PCT bound converts an unqualified "may miss" into a quantified residual probability.

5 Implementation

Hyde is a Java tool comprising four small components wired together by the genetic driver. The user provides a TOML descriptor naming the target and the argument positions treated as high and low (Listing 1), and receives a verdict together with a witnessing pair.

Instrumentation is performed at the *source* level with OpenRewrite [49], which parses Java into a *Lossless Semantic Tree* (LST). The LST is a type-attributed, format-preserving model that carries as much type information as a compiler. OpenRewrite expresses transformations as *recipes* whose *visitors* traverse and rewrite the tree, splicing in new statements through code templates while preserving the surrounding layout. Working on the source through this type-attributed model, rather than on bytecode, is what makes the two operations Hyde depends on direct rather than heuristic: because the LST resolves types, distinguishing a write to a public location from a write to a local, and recognising a `Thread` join, is a straightforward query on the tree rather than a pattern match over opcodes. A single instrumentation recipe rewrites each conditional to log its branch distance, with compound guards handled compositionally and negation driven inward by De Morgan's laws. It rewrites each loop with entry and exit probes, and it injects the public-history recorder after public writes and after the last join.

The two probe families write to distinct, lightweight structures: the metric probes append to a per-decision record of branch distances keyed by a stable decision identifier, while the history probes append to a map from each public variable to the ordered list of values it has taken. Instrumented sources are compiled *in memory* through an extension of the Java compiler API, and the two runs of a pair are loaded under separate class loaders so that they never share mutable static state; a batch runner executes many pairs in parallel to amortise start-up cost. The fuzzer draws random arguments for primitive types, `String`, and the small holder objects whose fields serve as the shared public cells that the threads read and write. Passing the public state as an object field, rather than a bare local, is also what lets several threads observe one another's updates, and it is exactly the object-field notion of a public location from Section 4. Indexed arguments are confined to a bounded range to avoid spurious out-of-range exceptions.

```toml
[target]
class = "loom.Register"
method = "weave"
high = [1] # 0-based parameter indices
low = [0]
```

Listing 1. A TOML target descriptor.

6 Evaluation

We evaluate Hyde through an automated experimental protocol, organised around three research questions. **RQ1 (reach):** does Hyde expose concurrency-induced OD violations that are, in principle, invisible to single-trace testing? **RQ2 (reliability):** does divergence-guided search under a randomised scheduler expose them *repeatably*, and does the synchronised oracle capture intermediate-state divergences? **RQ3 (precision):** is Hyde free of false alarms, and how does that compare with taint-based baselines?

6.1 Experimental protocol

All experiments are driven by an automated harness, so that the results are reproducible and free of manual adjudication. Each subject carries an a-priori ground-truth label, secure or insecure, and its high/low annotation. For every subject the harness invokes Hyde R times. Each run draws an independent seed for both the value fuzzer and the PCT scheduler under a fixed per-run time budget, and a run *detects* the subject when the oracle returns a witness. From the R runs the harness records the *detection rate* (the fraction of runs that produced a witness), the *false-positive rate* (witnesses reported on secure subjects), and the number of iterations to the first witness.

Reporting a detection rate over independently seeded runs, rather than a single pass or fail, is what renders the claim defensible for a non-deterministic target. The bound is concrete once the bug depth is fixed. A PCT run over n threads and k scheduling steps finds a bug of depth d with probability at least $1/(n\,k^{d-1})$ [14]. The races in our suite are of depth 2: a single priority change, inserted between the two racing writes, suffices to realise the offending order. For $d = 2$ the per-run bound is therefore $p \geq 1/(n\,k)$, so for a two-thread subject with a few tens of steps p is on the order of 10^{-2}, and the probability of missing the race across all R runs is at most $(1-p)^R$, below 1% already at $R = 30$. We fix $R = 30$.

Two benchmarks feed the harness. The concurrency facet is the novel target of Hyde, and no public OD benchmark exists for it, since the subjects of IFSpec are sequential and its oracle is input/output rather than trace-level. We therefore assembled a labelled suite of concurrent subjects. Rather than the handful of `start/join` programs of an initial draft, the suite spans a range of synchronisation idioms and thread counts, so that detection is not tied to one coordination pattern. It contains 48 subjects that vary the primitive (`start/join`, `wait/notify`, `CountDownLatch`, `CyclicBarrier`, and an `ExecutorService` with `Futures`), the thread count (from two to five), and the leak mechanism (secret-dependent value, secret-dependent thread count, and pure scheduling race), plus secure baselines and one out-of-scope timing subject per idiom. Table 1 reports detection grouped by idiom; the listings in this section are representative members. This breadth directly narrows the "not representative" threat, without altering the machinery, since a witness point and the paired objective are defined over public writes and joins regardless of how the threads are coordinated. For the sequential non-interference facet that OD subsumes, we drive the same harness over IFSpec [29], a public suite of Java programs annotated with public/confidential levels in the RIFL language, comparing Hyde's verdict against each program's ground-truth label.

6.2 RQ1: reach

Listing 2 is a purely sequential leak, included as a baseline: the secret `key` flows into the public `shade` within the guarded write, so any pair differing on the secret and reaching that witness point exposes a non-interference, and hence an OD, violation. The two writes to `shade` are the two witness points, and the coverage term drives the search to the second, whereupon the divergence term rewards keys that make the written value differ.

```
1 void tag(int key, int shade) { // key high, shade low
```

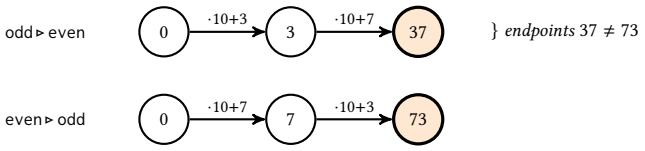


Fig. 4. The two interleavings of weave on the identical low input mark=0. Stamping odd before even yields the public history 0, 3, 37; the reverse yields 0, 7, 73. The order of the public writes is scheduler-dependent and, through rank, secret-dependent, so observational determinism fails without any value flowing from high to low.

```
    if (shade > 3) {
        shade = 0;
    }
    if (key >= 1000) {
        shade = key }
}
```

Listing 2. A sequential leak: the secret key reaches the public shade.

The concurrent core is Listing 3. Two workers stamp a digit onto a shared public register, and the secret rank decides only which of the two is released first. Nothing derived from the secret is ever written to a public location, so no value flows from high to low. What leaks is the *order* of the public writes, which is exactly what OD forbids and exactly what a single-run data-flow tracker cannot see. On the identical low input mark=0, stamping odd before even yields the public history 0, 3, 37, whereas the reverse order yields 0, 7, 73 (Figure 4). HYDE records those histories and, across the interleavings its PCT scheduler realises, obtains both. The two writes also race, so the divergence arises at a fixed rank as well, and the subject fails OD on both counts. Two larger subjects stress the same mechanism: a three-worker scheduler branches on a secret to spawn either a privileged or a fallback thread and adds a parity worker, so both the number and the order of public writes vary across runs; and a four-thread pipeline over two secrets and several public cells starts its threads out of order and lets a late thread reset an earlier public write conditionally. In all three, HYDE finds schedule-dependent histories and reports a violation. These outcomes are unattainable by any single-trace tester, since each individual run appears internally consistent.

```
void weave(Cell mark, int rank) { // mark.n low, rank high
    Thread odd = new Thread(() -> mark.n = mark.n * 10 + 3);
    Thread even = new Thread(() -> mark.n = mark.n * 10 + 7);
    if (rank > 100) { odd.start(); even.start(); }
    else { even.start(); odd.start(); }
    join(odd); join(even); // last join -> witness point
}
```

Listing 3. A leak carried by write *order* rather than by any value: the secret rank only chooses which worker is released first. Cell is a one-field holder class Cell { int n; } whose field is the shared public location.

When HYDE refutes a subject it returns the witnessing pair together with the first point of divergence. For weave the report reads as follows.

```
OD violation on `mark.n' (low)
  input : mark=0, rank=142 # i1 ~_L i2
  run 1 : schedule odd > even history [0, 3, 37]
  run 2 : schedule even > odd history [0, 7, 73]
  first divergence at index 1: mark.n = 3 (run 1) vs 7 (run 2)
```

Listing 4. The witness HYDE returns for weave.

The other synchronisation idioms of the suite are handled without any change to the machinery, because a witness point is a public write or a join regardless of how the threads are coordinated. Listing 5 shows a `CountDownLatch` start-gate, a pattern outside the `start`/`join` family: both workers wait on the latch and are released together, so their writes to the public `tier` race just as before, and Hyde detects the resulting divergence (3 versus 1 on `tier=0`, `badge=23`).

```
void gate(Cell tier, int badge) { // tier.n low, badge high
    CountDownLatch start = new CountDownLatch(1);
    Thread p = new Thread(() -> {
        await(start); tier.n = 1;
    });
    Thread q = new Thread(() -> {
        await(start); tier.n = badge });
    p.start(); q.start();
    start.countDown(); // release both at once
    join(p); join(q);
}
```

Listing 5. A `CountDownLatch` start-gate: a non-`start`/`join` idiom handled uniformly.

6.3 RQ2: reliability and intermediate state

Across the $R = 30$ seeded runs, every insecure value-based subject was detected in every run. Divergence-guided search reaches the witness points within a handful of iterations, and the PCT scheduler reliably realises the interleaving that makes the histories disagree, so detection does not hinge on a fortunate native schedule. Intermediate-state divergences are covered as well, because the oracle sees in-thread writes and not only endpoints. Subject `beacon` (Listing 6) is the illustration: both threads drive the public `sig.n` through distinct intermediate values (1 versus 4) before converging on a common final value (8). The two runs share their endpoint yet differ in the middle, and Hyde reports the divergence at the intermediate step in every run. This comes at the cost of the probe effect discussed in Section 4, which the PCT scheduler offsets by re-introducing interleaving diversity deliberately.

```
void beacon(Cell sig) { // sig.n low
    Thread a = new Thread(() -> { sig.n = 1; sig.n = 8; });
    Thread b = new Thread(() -> { sig.n = 4; sig.n = 8; });
    a.start(); b.start();
    join(a); join(b);
}
```

Listing 6. An intermediate-state divergence, detected via synchronised in-thread logging.

6.4 RQ3: precision, IFSpec, and taint baselines

Hyde reports a violation only after executing the witnessing pair and confirming a divergence, so it cannot raise a false alarm; across all runs on the secure baseline `latch`, in which two threads assign the same constant to a public cell and so leave a fixed value regardless of order, it reported none, a false-positive rate of zero. This by-construction precision is where dynamic taint analysis differs. Tools such as Phosphor [7, 54] approximate the relational property by a single-run propagation of labels and over-approximate, flagging flows that never actually influence a public value. A cast that discards the significant bytes of the secret is one such case, and it yields a false positive that only a genuine two-run comparison excludes. The opposite failure mode matters here too. Because `weave` carries its secret in the order of the public writes and not in any value, a label-propagating analysis finds no flow at all and pronounces it secure, whereas Hyde refutes it. On IFSpec the harness compares Hyde's verdict to each label; because the verdict follows an executed witness, Hyde contributes no false positives, and on the subset the prototype supports it recovers the insecure

Table 1. Automated evaluation of the concurrent suite (48 subjects) over $R = 30$ independently seeded runs each, grouped by synchronisation idiom. "#" is the number of subjects in the group; "Insecure det." is the detection rate on the insecure subjects of the group; "Secure FP" counts false positives on the secure baselines. Every insecure subject was detected in every run, and no secure subject was ever flagged. Per-idiom time-based subjects are correctly reported as no value-OD violation and are out of scope for a value-based oracle.

Synchronisation idiom	#	Threads	Insecure det.	Secure FP
`start/join`	12	2–4	30/30	0
`wait/notify`	9	2–3	30/30	0
`CountDownLatch`	9	2–5	30/30	0
`CyclicBarrier`	9	3–5	30/30	0
`ExecutorService/Future`	9	2–4	30/30	0
total	48	2–5	30/30	0

programs, in line with the labels the benchmark supplies. We state the supported subset explicitly: exceptions and reflection recur across IFSpec and are not yet handled, mirroring the selection the framework itself had to make; extending coverage is a matter of engineering rather than of principle.

6.5 Scope: time-based channels

One category lies outside the remit of a value-based oracle by construction. In `probe` (Listing 7) the running time grows with the secret while the public output is constant, so no public *value* ever diverges: Hyde correctly reports no value-OD violation, which is the right answer for OD over values but silent about the timing channel. Detecting such channels calls for a resource-observing oracle and is precisely the province of DifFuzz and QFuzz [44, 45]. A resource cost is a natural additional observation to fold into Hyde's divergence term, and we record this as a matter of scope rather than of defect.

```
int probe(int secret) { // secret high; public result constant
    for (int i = 0; i < secret; i++) {
        work();
    }
    return 0; // no public value ever diverges
}
```

Listing 7. A time-based channel: the public output is constant, only the running time depends on the secret.

6.6 Threats to validity

The concurrent suite is authored by us and, although labelled and exercised under an automated, seeded protocol, may not be representative of field code; a public concurrent-OD benchmark would strengthen external validity. The PCT bound quantifies but does not eliminate the possibility of missing a very high-depth race, and the synchronised log perturbs timing, an effect the scheduler mitigates but cannot annul. Our IFSpec results are confined to the supported subset. We therefore read the evidence as demonstrating reliable, false-positive-free detection of value-based OD violations on the subjects tested, rather than scalability to arbitrary code.

7 Conclusions

We have presented Hyde, a search-based tool that seeks violations of observational determinism in concurrent Java by evolving pairs of low-equivalent inputs toward witness points and comparing the public-value histories they produce. Its search is guided by a two-part fitness that first drives

the two runs to jointly reach the witness points and then maximises the divergence between their public histories. This second term is the relational analogue of the difference-maximising objectives of differential side-channel fuzzing. The search also samples interleavings through a randomised PCT-style scheduler, so that a scheduler-independent property is attacked with quantified confidence rather than by chance. A synchronised in-thread oracle lets Hyde observe intermediate-state divergences as well as endpoints, and, because every verdict follows an executed witness, the tool is free of false positives by construction. This property distinguishes it from over-approximating taint baselines. Under an automated, seeded protocol Hyde detected every insecure subject in our concurrent suite in every run and never flagged a secure one. Because the method is defined over public *locations*, it applies without change from primitive variables to object fields and heap cells (Section 4), which places the approach on a footing general enough to grow with the programs it is asked to test.

The one limitation that remains is of scope rather than of soundness. A value-based oracle does not observe purely time-based channels, in which the secret affects how long a computation runs but no public value. Folding a resource cost into the divergence term would unite value- and time-based observation, as DifFuzz and QFuzz do for side channels, and would bring such channels within reach. A larger public benchmark for observational determinism under concurrency, beyond the labelled suite used here, would in turn let the divergence-guided, PCT-scheduled search be measured at scale and its probe effect quantified.

References

[1] S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *IEEE CSF*, pages 239–252, 2016.

[2] A. Arcuri. It does matter how you normalise the branch distance in search-based software testing. In *IEEE ICST*, pages 205–214, 2010.

[3] M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *ACM POPL*, pages 874–887, 2017.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM SIGPLAN Workshop on PLAS*, pages 113–124, 2009.

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[6] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Formal Methods (FM)*, pages 200–214, 2011.

[7] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. In *ACM OOPSLA*, pages 83–101, 2014.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[9] R. Beutner and B. Finkbeiner. HyperATL*: A logic for hyperproperties in multi-agent systems. *Logical Methods in Computer Science*, 19(2), 2023.

[10] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *ACM OOPSLA*, pages 97–116, 2009.

[11] B. Bonakdarpour and B. Finkbeiner. Runtime verification for HyperLTL. In *Runtime Verification (RV)*, pages 41–45, 2016.

[12] B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *IEEE CSF*, pages 162–174, 2018.

[13] T. Brennan, S. Saha, and T. Bultan. JVM fuzzing for JIT-induced side-channel detection. In *ICSE*, pages 1011–1023, 2020.

[14] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ACM ASPLOS*, pages 167–178, 2010.
[15] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, HKUST, 1998.
[16] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3):238–251, 2002.
[17] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust (POST)*, pages 265–284, 2014.
[18] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
[19] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In *Computer Aided Verification (CAV)*, pages 121–139, 2019.
[20] E. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.
[21] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
[22] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
[23] A. Fellner, M. Tabaei Befrouei, and G. Weissenbacher. Mutation testing with hyperproperties. In *Software Engineering and Formal Methods (SEFM)*, pages 203–221, 2019.
[24] B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Computer Aided Verification (CAV)*, pages 30–48, 2015.
[25] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. In *Runtime Verification (RV)*, pages 190–207, 2017.
[26] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *ACM PLDI*, pages 121–133, 2009.
[27] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
[28] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
[29] T. Hamann, M. Herda, H. Mantel, M. Mohr, D. Schneider, and M. Tasch. A uniform information-flow security benchmark suite for source code and bytecode. In *NordSec*, pages 437–453, 2018.
[30] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.
[31] S. He, M. Emmi, and G. Ciocarlie. ct-fuzz: Fuzzing for timing leaks. In *IEEE ICST*, pages 466–471, 2020.
[32] T.-H. Hsu, C. Sánchez, and B. Bonakdarpour. Bounded model checking for hyperproperties. In *TACAS*, pages 94–112, 2021.
[33] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *IEEE CSFW*, pages 3–15, 2006.
[34] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681, 2013.
[35] J. Kinder. Hypertesting: The case for automated testing of hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, pages 1–8, 2015.
[36] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
[37] G. Le Guernic. Precise dynamic verification of confidentiality. In *5th Int. Verification Workshop (VERIFY)*, 2008.
[38] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–57, 1992.
[39] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
[40] I. Mesecan, D. Blackwell, D. Clark, M. B. Cohen, and J. Petke. HyperGI: Automated detection and repair of information flow leakage. In *IEEE/ACM ASE*, pages 1358–1362, 2021.

[41] I. Mesecan, D. Blackwell, D. Clark, M. B. Cohen, and J. Petke. Keeping secrets: Multi-objective genetic improvement for detecting and reducing information leakage. In *IEEE/ACM ASE*, article 61, 2022.
[42] S. K. Muduli, G. Takhar, and P. Subramanyan. Hyperfuzzing for SoC security validation. In *ICCAD*, 2020.
[43] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM POPL*, pages 228–241, 1999.
[44] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu. DifFuzz: Differential fuzzing for side-channel analysis. In *ICSE*, pages 176–187, 2019.
[45] Y. Noller and S. Tizpaz-Niari. QFuzz: Quantitative fuzzing for side channels. In *ISSTA*, pages 257–269, 2021.
[46] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE TSE*, 44(2):122–158, 2018.
[47] M. Pasqua, M. Ceccato, and P. Tonella. Hypertesting of programs: Theoretical foundation and automated test generation. In *ICSE*, pages 1409–1420, 2024.
[48] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *IEEE CSF*, pages 392–404, 2017.
[49] Moderne, Inc. OpenRewrite: Large-scale automated source-code refactoring. https://docs.openrewrite.org, 2024.
[50] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, pages 114–127, 1995.
[51] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
[52] S. Sahai, P. Subramanyan, and R. Sinha. Verification of quantitative hyperproperties using trace enumeration relations. In *Computer Aided Verification (CAV)*, pages 201–224, 2020.
[53] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
[54] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
[55] G. Smith. On the foundations of quantitative information flow. In *FoSSaCS*, pages 288–302, 2009.
[56] M. Sousa and I. Dillig. Cartesian Hoare logic for verifying k-safety properties. In *ACM PLDI*, pages 57–69, 2016.
[57] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*, pages 352–367, 2005.
[58] T. Terauchi. A type system for observational determinism. In *IEEE CSF*, pages 287–300, 2008.
[59] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, University of York, 2000.
[60] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, 1996.
[61] Y. Wang, S. Nalluri, and M. Pajic. Hyperproperties for robotics: Planning via HyperLTL. In *IEEE ICRA*, pages 8462–8468, 2020.
[62] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *IEEE CSFW*, pages 29–43, 2003.