Exploring the Design Space of Isolation Mechanisms

Sidhartha Agrawal
University of British Columbia, Canada

Shaurya Patel
University of British Columbia, Canada

Linh Pham
Hammerspace, USA

Arya Stevinson
Oracle, Canada

Ilias Karimalis
University of British Columbia, Canada

Hugo Lefeuvre
University of British Columbia, Canada

Aastha Mehta
University of British Columbia, Canada

Reto Achermann
Technical University of Munich, Germany

Margo I. Seltzer
University of British Columbia, Canada

Abstract

Processes, containers, and virtual machines (VMs) offer different trade-offs between performance, isolation, and resource sharing. Yet, they represent just three points in a vast design space. Developers repeatedly introduce ad-hoc isolation mechanisms to tune these trade-offs for specific deployments, but there is no systematic way to explore the design space and derive mechanisms that satisfy given security and performance goals.

We introduce *IsoSearch*, a technique for exploring the isolation mechanism design space to find one that satisfies the goals and constraints of a specific deployment scenario. *IsoSearch* builds on *OSmosis* [4], our prior model of isolation, which represents protection domains, resources, and their relationships as a graph. A developer begins with a starting mechanism and its representation as an *OSmosis* graph and specifies both a set of functional constraints (e.g., these two protection domains must both be able to write to this file) and an optimization target (e.g., minimize the amount of memory shared between the two domains). *IsoSearch* then navigates the design space by applying valid graph transitions to generate candidate mechanisms, scores each mechanism, and selects the most promising ones for further exploration. The search terminates when all active candidates satisfy the constraints and goals or when score improvement plateaus. We demonstrate the efficacy of our approach through five case studies, ranging from resource privatization to protecting a database's private key.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Security and privacy → Virtualization and security**.

This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2877-8/2026/09
https://doi.org/10.1145/3831586.3838145

Keywords: Isolation mechanisms, Design-space exploration, Beam search

ACM Reference Format:
Sidhartha Agrawal, Shaurya Patel, Linh Pham, Arya Stevinson, Ilias Karimalis, Hugo Lefeuvre, Aastha Mehta, Reto Achermann, and Margo I. Seltzer. 2026. Exploring the Design Space of Isolation Mechanisms. In *14th Workshop on Programming Languages and Operating Systems (PLOS '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831586.3838145

1 Introduction

Safely sharing computer systems among mutually distrusting users has been a long-standing challenge. Processes, containers, and virtual machines (VMs) represent three familiar points in a vast design-space of isolation mechanisms, each making different trade-offs between performance, isolation, and resource sharing. A key structural difference between these mechanisms is their *Trusted Computing Base* (TCB). A container shares the host OS kernel, expanding the TCB to all co-tenants, while a VM runs a dedicated kernel, bounding the TCB to the hypervisor. This TCB dimension is just one axis in the broader design space; even within a single mechanism (e.g., containers), the details of the TCB vary [4]. Recent work explores this space from multiple angles: hardening container isolation with stronger kernel boundaries [10, 21], offering lighter-weight hardware-enforced compartmentalization [17, 25, 27], and protecting sensitive data from co-located tenants via confidential VMs [18].

Selecting the right mechanism for a specific deployment remains challenging. Developers might need to use an existing mechanism with conservative isolation guarantees, while compromising on performance. Alternatively, developers might need to design a new custom isolation mechanism with precise security requirements, e.g., limit the set of resources that are shared between two instances of the mechanism. Designing custom isolation mechanisms is a tedious and error-prone task, as developers today state security requirements informally (e.g., "the key server must not be reachable from the network"), leaving their enforcement

implicit (e.g., which process should hold the key). Indeed, industry reports highlight misconfiguration as a leading source of security exposure [20], a direct consequence of this informality.

We show that isolation mechanisms can be viewed as points in a design space, characterized by security and performance metrics, that we can explore algorithmically. Given a starting configuration of an isolation mechanism, constraints describing the functionality of a target isolation mechanism, and one or more goals related to security (e.g., resources that must be isolated between two instances) and performance (e.g., minimizing the memory footprint of the isolation mechanism), we demonstrate that a search algorithm can derive new points in the design-space that satisfy the constraints and goals.

We present *IsoSearch*, a search procedure for deriving isolation mechanisms from user-specified functional constraints and goals.[1] *IsoSearch* leverages the *OSmosis* model [4], which represents a system as a graph of protection domains (e.g., processes, containers, VMs), resources, and their interactions. The search performs a series of *transitions* from one well-formed graph instance to another, with each transition corresponding to a concrete change in the properties of a deployable isolation mechanism.

Search does not operate in isolation. *IsoSearch* is the middle stage of a three-stage end-to-end pipeline (Fig. 1): LinTool [4], from our prior work, extracts an *OSmosis* graph from a running system (stage one); *IsoSearch*, the contribution of this paper, searches the design space for a graph that meets the developer's goals and constraints (stage two); and the discovered graph is instantiated on a target OS: a commodity system such as Linux or a capability-based OS such as CellulOS [3, 5] (stage three). We build this last stage for CellulOS, reporting a preliminary instantiation, and leave broader automation to future work (Sec. 6).

Contributions. We **(1)** identify twelve *OSmosis* graph *transitions* that produce the reachable design-space (Sec. 3.1); **(2)** define two new metrics that together define optimization goals, namely RSI (*Resource Similarity Index*), which quantifies pairwise resource sharing between protection domains, and *Memory Consumption (MC)* (Sec. 3.2); and **(3)** present the *IsoSearch* beam-search algorithm (Sec. 3.3). Through five case studies, we show that *IsoSearch* automatically derives isolation mechanisms from security goals and functional constraints. For example, our study shows how *IsoSearch*, applied to minimize the TCB of a PostgreSQL deployment, derives a dedicated TLS-key protection domain to mediate key accesses, a mechanism the developer never specified. We also show that, when goals do not determine a unique mechanism, *IsoSearch* exposes multiple valid alternatives.

[1] Our implementation is open source and available at https://github.com/ubc-systopia/plos2026-isosearch/commit/c91eef4.

Figure 1. The envisioned end-to-end pipeline. *IsoSearch* (this paper) is the middle stage, between LinTool [4] graph extraction and instantiation on a target OS (Sec. 6).

2 Primer on the *OSmosis* Model

The *OSmosis* model [4] is a graph-based representation of a system's isolation and sharing structure. A graph consists of three node types: *Protection Domains* (PDs), *Resources*, and *Resource Spaces* (allocation pools that group resources of the same type). Four types of edges connect the nodes: *Hold* (a PD owns or controls a resource or PD), *Request* (a PD may request resources from another PD), *Subset* (a resource belongs to its resource space), and *Map* (a mapping between resources or resource spaces). *Hold* edges carry permissions; *Request* edges carry a resource type.

A *OSmosis* graph is *well-formed* if it satisfies six structural invariants: 1) resources must be reachable from a PD via a *Hold* edge; 2) resources must be connected to a resource space of the same type via a *Subset* edge; 3) *Request* edges exist only between PDs; 4) there must be a resource space assigned to a *Request* edge; 5) the source of a *Hold* edge is a PD; and 6) *Map* edges connect nodes of the same type. *IsoSearch* must maintain these invariants during every transition. Some platforms (e.g., an OS kernel or hypervisor) further restrict the graph structure they can support; these can be encoded as additional constraints (see Sec. 6). Queries on the *OSmosis* graph report isolation metrics, e.g., the Trusted Computing Base (TCB) and Impact Boundary (IB) [4].

3 Design-Space Exploration

We use the *OSmosis* model to express the isolation and sharing profile of a system configuration and explore the design space by mutating the graph to achieve desired properties. Given a starting system configuration G_0 (i.e., a specific *OSmosis* graph), a set of functional constraints C (i.e., graph predicates the final configuration must satisfy), and an optimization goal O (based on metrics defined in Sec. 3.2), **find a *candidate* G' that satisfies C and minimizes/maximizes O**, where *candidate* refers to any well-formed *OSmosis* graph representing a point in the design space. The design space is too large to search exhaustively. Hence, *IsoSearch* makes the best effort to achieve O: it explores a bounded region guided by a scoring function (Sec. 3.3.1) but does not guarantee a globally optimal solution.

Model Element		Model Transition	Preconditions	Graph Operations(s)
Node	PD	Create	None	Add PD node
		Delete	No outgoing edges	Delete PD node
	Resource	Create	PD and resource space nodes exist	Add resource node, add hold & subset edges
		Delete	Only one hold edge	Delete resource node, delete hold & subset edges
	Resource Space	Create	PD node exists	Add resource space node, add hold edge
		Delete	No outgoing subset edges	Delete resource space node, delete hold edge
Edge	Hold	Add	PD and resource node exist	Add hold edge
		Delete	Not the last hold edge	Delete hold edge
	Request	Add	PD nodes exists	Add request edge
		Delete	None	Delete request edge
	Map	Add	Resource or resource space nodes exist	Add map edge
		Delete	No map edges between the resources of corresponding resource spaces	Delete map edge

Table 1. *OSmosis* model transitions, with the required preconditions for the transition. Some transitions involve more than one graph operation.

3.1 Design Space: Valid Graphs and Transitions

We first define what it means for an *OSmosis* graph to be valid (Sec. 3.1.1) and then introduce the transitions (Sec. 3.1.2) used to explore the design space.

3.1.1 Valid *OSmosis* Graphs. An *OSmosis* graph is valid if it is well-formed and satisfies functional constraints C. These constraints encode the operational requirements that allow the system to perform its task. We allow two forms of constraints that capture common isolation requirements, leaving more complex constraints for future work.

reachable$(PD_s, PD_d|Res, nEdges)$: A boolean predicate indicating whether PD_s can reach resource *Res* or PD_d via at most *nEdges* edges. A value of one for *nEdges* specifies direct access via a single *Hold* edge; larger values permit indirect access through intermediate PDs. This predicate captures access requirements naturally: "a given PD (PD_W) must be able to reach a specific resource (Res_{log})" becomes $reachable(PD_W, Res_{log}, \infty)$.

exists(PD): A boolean predicate indicating whether PD exists in the graph, ensuring that required application components remain present during exploration. *IsoSearch* only adds or splits PDs; it never removes a required one.

3.1.2 Transitions on the *OSmosis* Graph. We generate new candidates by applying *transitions* that correspond to semantically meaningful changes to the *OSmosis* graph, e.g., adding a PD or adding/removing a *Hold* edge. Starting from a fully-connected graph and deleting edges is one valid exploration strategy, but in practice we assume that G_0 represents an existing deployment and that the transitions model realistic modifications. We need both additions and removals to discover patterns that require structural reorganization (e.g., first adding a PD before transferring a *Hold* edge away from an existing one).

Let $wf(G)$ test whether graph G satisfies the structural invariants, and let a transition T be a graph operation with preconditions $PreCond_T$. Table 1 shows all transitions, their preconditions, and their corresponding graph operations. We construct each transition so that applying it to a well-formed graph G that satisfies $PreCond_T(G)$ again yields a well-formed graph. Let T_i be the i-th transition in a series of transitions during search, taking G_i to G_{i+1}. This gives the inductive step: $wf(G_i) \wedge PreCond_{T_i}(G_i) \wedge T_i(G_i) = G_{i+1} \implies wf(G_{i+1})$. Some transitions add/remove multiple edges/nodes to maintain well-formedness. For example, adding a new resource requires adding a *Hold* edge from a PD to a resource space and a *Subset* edge from the resource space to the resource. By induction, starting from a well-formed graph G_0, applying n transitions produces a well-formed graph G_n [2].

3.2 Evaluating Points in the Design Space: Metrics

Metrics are optimization targets that encode developer goals. We introduce two new metrics (RSI and MC) and use the established TCB metric [4] during design-space exploration. Users can define additional metrics as desired (e.g., the number of *Request* edges or the size of the privilege set), provided they are expressed as functions over the graph structure.

Resource Similarity Index (RSI): For two PDs holding resource sets R_i and R_j (optionally restricted to a single resource type), RSI is the Jaccard index $|R_i \cap R_j|/|R_i \cup R_j|$, or 0 if neither PD holds a resource. A lower RSI suggests stronger isolation. The global sharing index $\overline{\text{RSI}}$ is the mean RSI over all pairs of *resource-holding* PDs.

Memory Consumption (MC): The total memory footprint for the entire graph: the sum of all resource memory sizes (each resource counted once) plus a per-PD overhead for the PD's private memory state, such as its stack, heap, and runtime context.

Trusted Computing Base (TCB) [4]: For a single PD, identifies the set of PDs that a given PD must trust to function correctly. TCB (PD) includes (a) every PD reachable via *Request* edges from PD, and (b) every PD that co-holds a resource with PD directly, since either holder can corrupt the shared resource. A smaller TCB is generally preferable, so we assume that removing a PD from the TCB increases isolation. We use $|\text{TCB}(PD)|$, the cardinality of the trust set, as an optimization metric.

3.3 *IsoSearch* Algorithm

IsoSearch explores the design space using a *beam search* [24], a form of breadth-first search (BFS). The *beam* is the bounded-width set of highest-scoring candidate graphs that the search retains from one iteration to the next, and a *beam state* is a single graph within it (i.e., one *OSmosis* graph currently

[2] Mechanised in Verus by Achermann and Karimalis: https://github.com/ubc-systopia/osmosis-model.

being expanded). In each iteration, it generates candidates by applying every available transition to every eligible combination of nodes and edges in each beam state, narrowed only by the transitions' *preconditions*: no well-formedness violation, and no deletion of a PD, resource, or resource-space present in G_0: a constraint is trivially met once the object it names is deleted, which circumvents the requirement rather than meeting it. This preserves the components present in G_0: *IsoSearch* adds or splits PDs but never deletes initial state. No transition type is otherwise withheld. *IsoSearch* scores candidates (Sec. 3.3.1) and selects the top ones for the new beam; a candidate joins the result set when all constraints hold and every goal meets its threshold. *IsoSearch* stops when all beam states have found solutions or the mean beam score plateaus.

3.3.1 Scoring Function. A candidate's score has two components: constraint satisfaction S, which ranges over $[0, 5]$, and goal progress P, to which each goal contributes up to 10 points. Both ceilings are arbitrary: only their ratio, set by the weights α and β below, affects which graphs the search prefers.

Constraint satisfaction, $S \in [0, 5]$. S scales with the *fraction* of functional constraints a graph satisfies. A graph satisfying all of them receives the full score of 5, and one satisfying none receives 0. We score and explore candidates with violations to allow the search to pass through intermediate graphs that temporarily violate a constraint on the way to a solution.

Goal progress, P: For RSI and $\overline{\text{RSI}}$, progress is proportional to how much sharing has been reduced (or increased, for maximization): an RSI of 0 contributes the full 10 points, an RSI of 1 contributes 0.

For memory consumption (MC), progress scales linearly with the fraction of the budget remaining. For TCB, progress scales with how much the target PD's trust set has shrunk relative to the worst case of trusting all other PDs. At solution-validation time, we recompute all metrics to confirm that the candidate meets each goal's target threshold.

Figure 2. Exploration of mediation pattern: *IsoSearch* discovers a mediator PD enabling indirect access between PDs and resources, satisfying the specified goals and functional constraints. The sequence of transitions leading to a valid solution is shown in blue; discarded candidate in yellow.

Weight intuition. A candidate's final score is $W = \alpha S + \beta P$. We choose α/β so that any fully constraint-satisfying graph always outscores any constraint-violating graph, regardless of goal progress. Without this, the beam could improve a metric by dropping a required *Hold* edge, satisfying the goal but violating a constraint. Dominance bounds the ratio: each violation costs α while each of the $|G|$ goals adds at most 10 to P, so it requires $\alpha > 10\,\beta\,|G|$. With up to three goals we set $\alpha/\beta = 50$. To avoid overfitting weights, we ran *IsoSearch* on three representative scenarios: Mediation, Budget-Constrained Tenant Isolation, and Policy-Driven Privilege Separation, with $\alpha/\beta \in \{5, 10, 20, 50\}$. Across all twelve runs, the first-solution iteration and the core solution structure were identical, confirming that the specific weight values affect scoring magnitude but not the solutions discovered.

3.3.2 Example Walk-Through. In the following example, we use $\texttt{Res}\langle log \rangle$ to denote a resource of type *log* and $\texttt{Res}_{log}$ for a specific node of that type. Consider the scenario of webserver ($\texttt{PD}_W$) and database ($\texttt{PD}_{db}$) containers that use a log file ($\texttt{Res}_{log}$) to store events. When both containers use the same log file directly, they can both observe each other's events and corrupt the log file. Fig. 2 illustrates this setup and shows intermediate paths explored when trying to isolate the log.

Functional Constraints: C_1) $\texttt{PD}_W$ and $\texttt{PD}_{db}$ must reach $\texttt{R}_{log}$, and C_2) $\texttt{R}_{log}$ must have only one incoming *Hold* edge.

$$(C_1) \quad \mathit{reachable}(\texttt{PD}_W, \texttt{R}_{log}, \infty) \ \wedge \ \mathit{reachable}(\texttt{PD}_{db}, \texttt{R}_{log}, \infty)$$
$$(C_2) \quad \forall \texttt{PD}_i, \texttt{PD}_j \mid \mathit{reachable}(\texttt{PD}_i, \texttt{Res}_{log}, 1) \wedge \mathit{reachable}(\texttt{PD}_j, \texttt{Res}_{log}, 1) \implies i = j$$

Goal: To prioritize data integrity of logs, $\texttt{PD}_W$ and $\texttt{PD}_{db}$ must minimize sharing of log files.

$$O = \texttt{argmin}\ \mathit{RSI}(\texttt{PD}_W, \texttt{PD}_{db}, \texttt{Res}\langle log \rangle)$$

Exploration: *IsoSearch* reaches a solution in six steps, evaluating 2,439 candidates (see Table 2). The combinatorial space is large even for small graphs: at each iteration, this instantiation step causes the number of candidates to grow with the number of nodes, transitions, and beam states. We show only a representative subset of candidates in the figure. Each graph, identified by an [iterationNumber, beamIndex] pair, indicates the goals met, constraints satisfied, and the score assigned to it. In G_0, the value of RSI ($\texttt{PD}_W$, $\texttt{PD}_{db}$, $\texttt{Res}\langle log \rangle$) is 1, hence the goal is not satisfied. Similarly, of the two constraints, only C_1 is satisfied. As the goal is to reduce RSI, candidates whose resulting graphs have fewer shared *Hold* edges on $\texttt{Res}\langle log \rangle$ receive higher scores. We show three (of the eight) candidates in the first iteration: G_{1_a}, G_{1_b}, and G_{1_c}. G_{1_c} scores lowest (shown in yellow) and is dropped; the winning path (blue) proceeds through $G_{1_a} \rightarrow G_{2_d} \rightarrow G_{3_b} \rightarrow G_{4_c} \rightarrow G_{5_a} \rightarrow G_{6_b}$.

The scoring function approximates progress; it is not a perfect ranking. A single-path greedy search is brittle under such an approximation: one misstep at any iteration might

exclude a path to the solution. Beam search tolerates this imperfection by maintaining multiple paths simultaneously: the highest-scored candidate at a given iteration need not be the one that reaches a solution first. A full breadth-first search (BFS) is similarly not feasible, as the number of candidates scales polynomially with the number of nodes, edges, and transitions. *IsoSearch* orders candidates by score, not by metric value alone. This allows the search to recover from constraint violations, at the cost of occasionally non-monotonic metric paths. Restricting exploration to metric-monotone candidates would risk pruning solutions that require a temporary metric regression.

4 Exploration Case Study

We present five case studies. The example of Sec. 3.3.2 illustrated the mediation pattern; the first row of Table 2 shows its exploration statistics. The four case studies below evaluate *IsoSearch* across qualitatively different scenarios, demonstrating that *IsoSearch* efficiently finds deployment strategies that satisfy developer constraints.

Test Setup: We implement *IsoSearch* in Python, using the scoring function and weights as presented in Sec. 3.3.1 across all five case studies. For each case study, we define an initial graph G_0, functional constraints, and goals. G_0 is well-formed but satisfies neither the constraints nor the goals. For scenarios that use a memory budget, we assign each PD a fixed overhead representing its runtime process state. This is an approximation: real overhead varies by workload, but a scenario-specific estimate suffices to model the cost of adding more PDs than necessary. We then run *IsoSearch*, which terminates when either all beam states have found solutions, the score fails to improve for three consecutive iterations, or the beam search reaches a predefined depth bound. We used a beam width of 12; each iteration evaluates hundreds of candidates (see Table 2) before pruning, ensuring no promising direction is discarded prematurely.

4.1 Bidirectional RSI: Privatization and Sharing

Two scenarios confirm that *IsoSearch* handles RSI goals in both directions: *privatization*, which converts a shared resource to a private one (minimizing RSI), and consolidation, which merges resources into a shared pool (maximizing RSI). *IsoSearch* finds solutions to both constraints in two steps (Table 2). Both scenarios share the same functional constraint (both PDs must be able to reach the log resource) and differ only in their goal direction; we omit further details for space.

4.2 Budget-Constrained Tenant Isolation

This case study models a real threat in LLM serving: cross-tenant KV-cache leakage. Modern LLM inference servers (e.g., vLLM [15], TensorRT-LLM [19]) accelerate token generation by caching the attention key/value (KV) vectors computed from a client's prior tokens. Model weights, while potentially proprietary to the provider, are shared across all tenants and need not be isolated between them. The KV data, by contrast, encodes each client's private query history and must be kept isolated. Design alternatives range from a single server process holding one cache per client (weak isolation) to a separate process per client sharing the model (stronger isolation, higher memory cost); the right tradeoff depends on the memory budget and the number of clients. *IsoSearch* automatically searches for both the isolation structure and the assignment of clients and resources that satisfy the constraints and goals. We elaborate with an example.

Model: We assume three clients. G_0 totals 210 MB (100 MB model, three 20 MB KV-caches, 50 MB process overhead) against a 320 MB GPU budget. The design question: given this budget, how many tenant PDs should exist and how should resources be assigned?

Constraints and Goals: The key property desired between PDs of different clients is that no single PD may simultaneously hold two different clients' KV-caches. We instantiate one constraint per client-pair (three in total) to specify this property. Three further constraints require each tenant PD to hold the shared model directly; otherwise the cheapest way to reduce sharing is to strip the model from a tenant rather than isolate its cache. We also specify two goals: (i) a goal that minimizes the global sharing index $\overline{\text{RSI}}$ across all PD pairs, and (ii) a memory bound Mem $\leq$ 320 MB.

Results. *IsoSearch* reaches the first complete solution in eight steps, evaluating 2,798 candidates and discarding 2,672 (96% pruned by the beam), ultimately discovering 12 solutions in 3.1 s. This recovers the per-tenant isolation structure that directly eliminates the cross-tenant KV-cache side channel demonstrated by Wu et al. [30], without being told the answer in advance. The 12 solutions fall into two classes. Six are the intended design of Fig. 3 ($\overline{\text{RSI}} = 1/3$); the other six score *better*, $\overline{\text{RSI}} = 5/18$, by giving one PD a newly created 1 KB resource: a resource nobody shares dilutes the mean

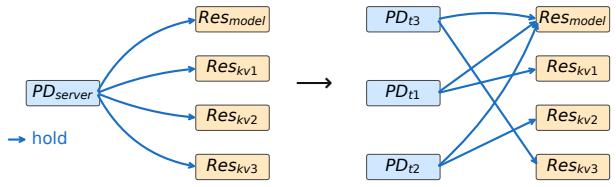

Figure 3. *IsoSearch* discovers the optimal tenant isolation structure for a multi-tenant ML inference server. G_0 (left): a monolithic server holds the shared model and all three per-tenant KV-caches. G_n (right): three PDs each hold the shared model and one private KV-cache, giving $\overline{\text{RSI}} = 1/3$. A fourth PD would require 360 MB, exceeding the 320 MB budget.

Case Study	Functional Constraints	Goals	# Steps/ Depth	# Eval.	# Disc.	# Sol.	Time (sec)
Mediation (Sec. 3.3.2)	(C1) $reachable(PD_W, R_{log}, \infty) \wedge reachable(PD_{db}, R_{log}, \infty)$ (C2) $\forall\ PD_i, PD_j \mid reachable(PD_i, \mathtt{Res}_{log}, 1) \wedge reachable(PD_j, \mathtt{Res}_{log}, 1) \implies i = j$	argmin RSI($\mathtt{PD}_W, \mathtt{PD}_{db}, \mathtt{Res}\langle log \rangle$)	6/10	2439	2339	8	2.2
Privatization (Sec. 4.1)	(C1) $reachable(PD_W, \mathtt{Res}_{log}, \infty) \wedge reachable(PD_{db}, \mathtt{Res}_{log}, \infty)$	argmin RSI($\mathtt{PD}_W, \mathtt{PD}_{db}, \mathtt{Res}\langle log \rangle$)	2/10	1246	1169	22	1.1
Sharing (Sec. 4.1)	(C1) $reachable(PD_W, \mathtt{Res}_{log}, \infty) \wedge reachable(PD_{db}, \mathtt{Res}_{log}, \infty)$	argmax RSI($\mathtt{PD}_W, \mathtt{PD}_{db}, \mathtt{Res}\langle log \rangle$)	2/10	4096	3983	37	6.0
Budget-Constrained Tenant Isolation (Sec. 4.2)	(C1) $\forall \mathtt{PD}_i,\ r \neq r' \in \{\mathtt{Res}_{kv1}, \mathtt{Res}_{kv2}, \mathtt{Res}_{kv3}\}$: $\neg(reachable(\mathtt{PD}_i, r, 1) \wedge reachable(\mathtt{PD}_i, r', 1))$ (C2) $\forall i \in \{1,2,3\}$: $reachable(\mathtt{PD}_i, \mathtt{Res}_{model}, 1)$	argmin $\overline{\text{RSI}}$ Mem $\leq$ 320 MB	8/12	2798	2672	12	3.1
Policy-Driven Privilege Separation (Sec. 4.3)	(C1) $\forall \mathtt{PD}_i,\ \forall x \neq \mathtt{Res}_{tls}$: $\neg(reachable(\mathtt{PD}_i, \mathtt{Res}_{tls}, 1) \wedge reachable(\mathtt{PD}_i, x, 1))$ (C2) $reachable(PD_{frontend}, \mathtt{Res}_{tls}, \infty) \wedge \neg reachable(PD_{frontend}, \mathtt{Res}_{tls}, 1)$ (C3) $reachable(PD_{frontend}, \mathtt{Res}_{auth}, \infty) \wedge \neg reachable(PD_{frontend}, \mathtt{Res}_{auth}, 1)$	argmin $\lvert\text{TCB}(PD_{frontend})\rvert$ argmin $\lvert\text{TCB}(PD_{backend})\rvert$ argmin $\overline{\text{RSI}}$ Mem $\leq$ 530 MB	6/25	10916	10616	109	20.5

Table 2. Summary of exploration statistics for five case studies. For each case study, we specify the functional constraints and goals, and report the number of candidates evaluated, candidates discarded (i.e., not included in a beam), wall-clock time, and solutions found in the entire exploration. *Steps* is the number of transitions applied to G_0 to reach the first complete solution; *Depth* is the depth bound given to the search (the algorithm continues after the first solution to accumulate additional ones). The functional constraint $exists(\mathtt{PD}_W) \wedge exists(\mathtt{PD}_{db})$ applies to the first three scenarios.

without isolating anything. The goal as stated rewards private resources, and at 1 KB the move is nearly free under MC; charging created resources a realistic size would price it out.

4.3 Policy-Driven Privilege Separation

As shown in Fig. 4, this case study combines three kinds of isolation constraints (direct-hold prohibition, co-location, and indirect-access) in a single secure database scenario. Consider a PostgreSQL-like database composed of a connection handler ($PD_{frontend}$), a storage manager ($PD_{backend}$), and an authentication subsystem (PD_{admin}), each with well-understood privilege boundaries. We wish to enforce three constraints: (C1) `tls_key` may not share a PD with any other resource, (C2) $PD_{frontend}$ must access `tls_key` only via a mediator, and (C3) $PD_{frontend}$ must access `auth_hba` only via a mediator.

Model. G_0 is inspired by PostgreSQL [29]. We model nine resources in three groups: connection resources (Res_{sock}, Res_{pipe}, Res_{tmp}) held by $PD_{frontend}$; storage resources (Res_{data}, Res_{wal}) held by $PD_{backend}$; and sensitive resources (Res_{tls}, Res_{auth}, Res_{log}, Res_{adm}), where Res_{tls} models the TLS private key and Res_{auth} models the host-based authentication config (`pg_hba.conf`): Res_{log} and Res_{adm} are held only by PD_{admin}; `auth_hba` is shared between $PD_{frontend}$ and PD_{admin}; and `tls_key` is shared between $PD_{frontend}$ and $PD_{backend}$ (over-provisioned; the solution removes it from $PD_{backend}$). Together, the three PD pairs yield $\overline{\text{RSI}} = \frac{1}{3}\left(\frac{1}{7} + \frac{1}{7} + 0\right) \approx 0.095$. Each of the first two pairs shares one resource over a seven-element union (frontend holds five, backend holds three, with `tls_key` as the sole overlap, giving $5 + 3 - 1 = 7$), so each contributes RSI = 1/7. The backend/admin pair shares nothing, contributing 0. A *Request* edge connects $PD_{frontend}$ to PD_{admin}. The nine resources total 350.02 MB (350 MB of bulk data plus 16 KB of keys and logs), and each PD incurs 30 MB of overhead, so G_0 costs 440.02 MB.

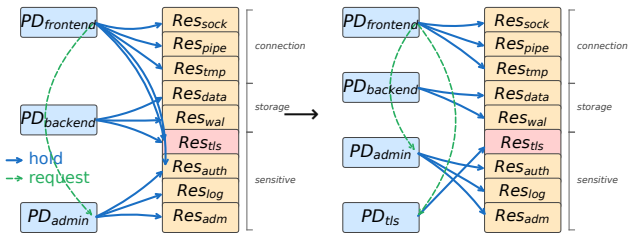

Figure 4. *IsoSearch* derives a trust-tiered design for a PostgreSQL-like deployment. G_0 (left): an over-provisioned 3-PD baseline where $PD_{frontend}$ directly holds `tls_key` and `auth_hba`, with `tls_key` shared with $PD_{backend}$. G_n (right): `tls_key` is isolated into a dedicated PD_{tls}, and $PD_{frontend}$ reaches sensitive resources only via *Request* edges (green, dashed) to PD_{admin} and PD_{tls} (mediation); *Hold* edges are blue. $|\text{TCB}(PD_{frontend})| = 2$ (both intentional REQUEST edges); $|\text{TCB}(PD_{backend})| = 0$.

Constraints and Goals. We encode C1 as eight co-location constraints, one per other resource, which force the dedicated PD_{tls} of Fig. 4. We encode C2 and C3 each as an indirect-access requirement: $PD_{frontend}$ must be able to reach the resource via some path but not hold it directly, forcing mediation. A Mem $\leq$ 530 MB bound allows at most five PDs: a sixth needs 530.02 MB, so those 16 KB put it over. We also specify three minimization goals ($\overline{\text{RSI}}$, $|\text{TCB}(PD_{frontend})|$, $|\text{TCB}(PD_{backend})|$), listed in Table 2. All constraints are violated at G_0. A manual approach could remove obvious violations one at a time, but cannot see that removing `tls_key` from $PD_{backend}$ creates a resource-held violation that must be resolved by introducing a new PD before the hold can be transferred, a dependency that requires reasoning about multi-step structural interactions. *IsoSearch* discovers this ordering automatically.

Results. *IsoSearch* reaches the first complete solution in six steps, evaluating 10,916 candidates and discarding 10,616 (97% pruned by the beam), ultimately discovering 109 solutions in 20.5 s. All 109 solutions contain the four-PD core structure of Fig. 4, at 470.02 MB. They vary in two respects:

(1) where unconstrained resources (Res_{log}, Res_{adm}) are assigned across the PDs, as constraints C1–C3 leave their placement unconstrained, and (2) whether the optional fifth PD is present. These alternatives are security-equivalent under the stated constraints but differ in ways that matter for operational concerns outside the current goal set; a manual approach would produce exactly one structure, while *IsoSearch* surfaces the full set of options and lets the operator choose based on criteria the policy did not anticipate or express.

5 Related Work

Design-space exploration for isolation mechanisms sits at the intersection of two lines of prior work: automated system optimization and isolation. Prior work automatically optimizes system memory footprint through kernel debloating [14] and configuration pruning [2, 23] and tunes runtime performance with causal inference [11], Bayesian optimization [13], neural networks [12], and LLMs [7, 9]. *IsoSearch* extends this tradition to the isolation design-space, where the goal is security rather than performance, and the search state is a formal graph model rather than a configuration vector.

The closest line of work in isolation is FlexOS [16], which presents a semi-automated exploration method for tuning a system's protection profile. *IsoSearch* is inspired by FlexOS' use of a partial order to compare configurations. However, unlike FlexOS, *IsoSearch* builds on a formal model (*OSmosis*), which provides greater expressiveness and flexibility in exploration and pruning, e.g., enabling arbitrary scoring functions and partial orders. Further, FlexOS' exhaustive search does not scale to large spaces, a problem that *IsoSearch* addresses with beam search.

Recent works surveying and systematizing isolation techniques provide a taxonomy of the design space [17, 28]. *IsoSearch* complements these works: where these works describe what has been built, *IsoSearch* automatically navigates the formal *OSmosis* design space to identify what *could* be built given a set of goals and constraints.

Finally, other works use static analysis to identify possible isolation strategies. μSCOPE [22] uses static analysis on program dependence graphs to identify over-privileged components in existing software, i.e., it determines where boundaries should be drawn to maximize least-privilege. *IsoSearch* addresses a complementary question: how could such a graph be refined to optimize towards different metrics or higher-level policy requirements? μSCOPE's output is a natural source for *IsoSearch*'s starting graph G_0. Other works, such as ACES [8] explore static clustering strategies to reduce the performance or memory cost of isolation. While such approaches might scale better than ours, they are fundamentally limited by the constraints and goals they can address.

6 Discussion & Future Work

The case studies validate *IsoSearch*'s core design; we now discuss limitations and future directions.

Completing the Pipeline: *IsoSearch* is the middle stage of the pipeline in Fig. 1: LinTool [4] supplies the input graph from a live system, and an end system is needed to instantiate *IsoSearch*'s output graph. We built this output stage in CellulOS [3, 5], an seL4-based OS that tracks *OSmosis* model state and exposes a flexible API for creating PDs, consuming an *OSmosis* graph directly. We validated the result by comparing CellulOS's own extracted graph against *IsoSearch*'s output. This instantiation allows us to ground the cost model: two designs that RSI rates identically (both $\overline{\text{RSI}} = 0$), a shared mediator PD versus two private ones, differ by **+9.7%** in cycles, the runtime cost of instantiating the additional protection domain and its private resources. Fully automating this stage across other backends such as Genode [1], FlexOS [16], CubicleOS [26], and Tyche [6], encoding each platform's structural restrictions on the graphs it can instantiate as additional *IsoSearch* constraints, remains future work.

Metrics and Intent: No single metric captures every notion of isolation. RSI ignores PD size and count and is blind to graph structure, so two topologies can score identically yet differ in trust: a shared mediator and a privatized one both yield $\overline{\text{RSI}} = 0$, but only the shared design forces both clients to trust a common ancestor PD. A goal can likewise be met while missing intent: minimizing $\overline{\text{RSI}}$ in the tenant scenario (Sec. 4.2) would collapse the shared model into one PD unless a constraint pins it. Choosing and combining metrics that capture complementary notions of isolation remains open.

Specifying Constraints From Natural Language: We currently author constraints manually, but an LLM could translate natural-language security policies, e.g., "the network handler must never hold the host key", directly into `reachable` predicates such as direct-hold prohibitions and indirect-access requirements (Sec. 3.1.1).

Information-Flow Isolation: The *OSmosis* model explicitly omits information-flow tracking [4]; extending it with taint labels would let *IsoSearch* additionally search for policies that prevent information leakage.

Code Content of Derived PDs: When *IsoSearch* creates a new PD, it specifies resource footprint and access structure but not executable content. For well-understood patterns (e.g., a mediator PD), the structure is sufficient guidance; for novel patterns, program synthesis or LLM-assisted code generation are promising directions for automating this process.

Acknowledgments

The authors would like to thank the anonymous reviewers of PLOS for their feedback on the draft. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] 2021. *Genode Operating System Framework.* https://genode.org/

[2] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. 2019. *Learning very large configuration spaces: What matters for Linux kernel sizes.* Ph. D. Dissertation. Inria Rennes-Bretagne Atlantique.

[3] Sidhartha Agrawal, Aastha Mehta, Arya Stevinson, Ethan Xu, Hugo Lefeuvre, Linh Pham, Margo Seltzer, Reto Achermann, and Shaurya Patel. 2025. CellulOS: An OS for Comparing Isolation Mechanisms. Talk, seL4 Summit 2025. https://sel4summit2025.sched.com/event/26GFQ/

[4] Sidhartha Agrawal, Shaurya Patel, Arya Stevinson, Linh Pham, Ilias Karimalis, Hugo Lefeuvre, Aastha Mehta, Reto Achermann, and Margo I. Seltzer. 2025. Comparing Isolation Mechanisms with OSmosis. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems* (Seoul, Republic of Korea) *(PLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/3764860.3768325

[5] Sidhartha Agrawal, Arya Stevinson, and Linh Pham. 2024. *CellulOS: An OSmosis-based OS personality on seL4.* https://cellulosdocs.readthedocs.io/ University of British Columbia, Systopia Lab.

[6] Charly Castes, Adrien Ghosn, Neelu S Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating Trust by Abolishing Hierarchies. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*. 231–238. doi:10.1145/3593856.3595900

[7] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, Qi Guo, and Yunji Chen. 2024. AutoOS: Make Your OS More Powerful by Exploiting Large Language Models. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*. doi:10.5555/3692070.3692362

[8] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Baltimore, MD, 65–82.

[9] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. 2025. TUNA: Tuning Unstable and Noisy Cloud Applications. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 954–973. doi:10.1145/3689031.3717480

[10] gVisor Authors. 2025. gVisor: The Container Security Platform. https://gvisor.dev/. [Accessed 07-07-2025].

[11] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: reasoning about configurable system performance through the lens of causality. In *Proceedings of the 17th European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 199–217. doi:10.1145/3492321.3519575

[12] Alexander Jung, Cezar Crăciunoiu, Nikolaos Karaolidis, Hugo Lefeuvre, Daniel Oñoro Rubio, Felipe Huici, Charalampos Rotsos, and Pierre Olivier. 2026. Wayfinder: Automated Operating System Specialization. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EuroSys'26)*. Association for Computing Machinery, New York, NY, USA, 710–727. doi:10.1145/3767295.3803589

[13] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro Rubio, Felipe Huici, and Mathias Niepert. 2021. Wayfinder: towards automatically deriving optimal OS configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 115–122. doi:10.1145/3476886.3477506

[14] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS'13)*.

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. ACM, 611–626. doi:10.1145/3600006.3613165

[16] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne Switzerland) *(ASPLOS'22)*. ACM, 467–482. doi:10.1145/3503222.3507759

[17] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P'25)*. IEEE Computer Society, Los Alamitos, CA, USA. doi:10.1109/SP61157.2025.00075

[18] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 8, 3 (2024), 1–42. doi:10.1145/3700418

[19] NVIDIA. 2024. TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM. Paged KV cache and KV-cache reuse are documented in `docs/source/advanced/kv-cache-reuse.md`.

[20] Qualys. 2024. *The State of Cloud and SaaS Security Report.* https://cdn2.qualys.com/docs/mktg/qualys-state-of-cloud-and-saas-security-report.pdf

[21] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. doi:10.1109/IOTSMS48152.2019.8939164

[22] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID'21)*. Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3471621.3471839

[23] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. 2014. Automatic feature selection in large-scale system-software product lines. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences* (Västerås, Sweden) *(GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 39–48. doi:10.1145/2658761.2658767

[24] Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

[25] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2022. CAP-VMs: Capability-Based Isolation and Sharing in the Cloud. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, Carlsbad, CA, 597–612. https://www.usenix.org/conference/osdi22/presentation/sartakov

[26] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems (Virtual, USA) *(ASPLOS'21)*. Association for Computing Machinery, New York, NY, USA, 546–558. doi:10.1145/3445814.3446731

[27] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. https://www.usenix.org/conference/atc20/presentation/shillaker

[28] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A Study of Security Isolation Techniques. *Comput. Surveys* 49, 3, Article 50 (oct 2016), 37 pages. doi:10.1145/2988545

[29] The PostgreSQL Global Development Group. 2023. PostgreSQL 16 Documentation. https://www.postgresql.org/docs/16/

[30] Guanlong Wu, Zheng Zhang, Yao Zhang, Weili Wang, Jianyu Niu, Ye Wu, and Yinqian Zhang. 2025. I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving. In *32nd Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2025.241772