Technical Report

SADRA:
A Sound Capability-based Access-Control
System for Disaggregated-Resource
Architectures

Contents

List of Tables

1 Introduction

Resource disaggregation strives to optimize resource utilization and improve elasticity by decoupling existing resources, such as CPUs, memories, and accelerators, from *processing elements* (PE) and distributing them into pools of heterogeneous resources, which PEs can access via fast mediums [1, 2, 3, 4, 5, 6]. Driven by the growing resource demands, disaggregated-resource architectures have recently attracted considerable attention from academia and industry [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. For example, various architectures and applications, such as *data centers* [18, 19, 20], *blade servers* [21], *rack-scale systems* [22, 23, 24], *cloud frameworks* [25, 26, 27], and *high-performance computing* [28, 29], have employed resource disaggregation to enhance their efficiency and performance.

Despite optimizing resource utilization, the distributed nature of disaggregated resources raises concerns about controlling access to them, such as: how an architecture provides fine-grained resource allocation for distributed resources; how it prevents processes from accessing unauthorized resources; and, how processes can delegate their access rights or revoke shared privileges. Furthermore, some disaggregated-resource architectures leverage *one-sided* communication—they replace processor/-software stacks on resource sides with hardware-based controllers— to improve performance [30, 31, 14, 1]. These controllers receive requests and directly read/write from/to resources based on them, thus raising security and privacy concerns [32].

Supporting elasticity introduces a new challenge to access-control systems. Elasticity in disaggregated-resource architectures means the ability and flexibility to scale [33, 34, 35]—resources and PEs should be able to attach to an architecture with minimum adaptation. For example, consider *Memory-Driven Computing* (MDC) [36], an abstract disaggregated-memory architecture comprising a pool of byte-addressable *Non-Volatile Memory* (NVM) modules and a fast-speed fabric. The architecture enables diverse PEs, such as CPUs and FPGAs, to attach the fabric and use the shared memory for storing data and other purposes, such as communicating channels. Nevertheless, an adversary process can access other processes' data by knowing data references. Thus, any suitable access-control system should support newly attached resources and PEs but cannot rely on PEs to control access. MDC demonstrates why a scalable and fine-grained access-control system is critical for these architectures; it provides direct and indirect access to a distributed and byte-addressable persistent shared memory for different PE types.

Recent works show that the *capability-based access-control* model [37, 38, 39] best suits disaggregated-resource architectures [40, 41]. The capability-based model employs unforgeable tokens, namely *capabilities*; thus, it can address various resource types, such as memory or computing resources. Furthermore, it provides a scalable and fine-grained controlling mechanism [40]. Moreover, it supports the *least-privilege principle* [42], thereby allowing fine-grained delegating or revoking access privileges.

In this paper, we first conduct a comparative analysis of capability systems because researchers have proposed several capability systems for different architectures [43, 44, 45, 46, 47, 48, 39, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 41]. We systematically reviewed existing capability systems to investigate which suit the architectures best. While reviewing capability systems, we extracted six traits from them, including dis-

tributed structure, subject and object granularity, capability delegation and revocation, and persistency (**??**). These traits enable us to reason about the capability systems' efficacy, performance, and applicability regarding disaggregated-resource architectures. For example, non-distributed capability systems, such as *CHERI* [48], do not suit these architectures because the distributed trait is a fundamental demand. Furthermore, we checked if they require HW/SW support.

Our investigation demonstrates that none of the existing capability systems covers all the extracted traits, and implementing their designs may be unpragmatic. For example, consider *CEP* [58] and *SemperOS* [41], two capability systems that supports most traits. CEP only involves resource-side controllers, thus introducing performance overheads due to communicating with PE-side processes. SemeprOS expects PE manufacturers to implement and integrate its suggested method into their products, which renders its approach impractical. Because access control constructs the basis for any secure or privacy-preserving framework and none of the systems supports all the traits, we decided to design a suitable access-control system for disaggregated-resource architectures.

In the second part of this paper, we present SADRA, a general-purpose capability-based access-control system for disaggregated-resource architectures that supports all the above traits. Making a general-purpose system enables architectures to adapt and employ it according to their requirements. For example, they can adapt our system to support direct or indirect memory access. Furthermore, the distributed nature of our access-control system enables other architectures, such as *Grid computing* [59] and *GPU clustering* [60], to employ it to control access to their shared distributed resources. SADRA provides efficacy, performance, and applicability by controlling access as close to processes as possible and providing fast capability-hierarchies revocation. Moreover, it resolves the one-sided communication challenge using co-processors, eliminating the need for processor/software stacks.

Our system's design controls access at two stages using controllers at PE-sides and resource-sides (Section 4). PE-side controllers enable the system to start controlling access close to processes, reducing the workloads of resource-side controllers. In addition, PE-side controllers can handle internal delegation requests without involving other controllers. Resource-side controllers perform final controls before granting access to resources. Furthermore, they manage inter-PE delegations, which enables them to revoke capability hierarchies efficiently; thus, PE-side controllers never communicate with each other. Consequently, the above three characteristics make our access-control system a scalable one.

We verify that our capability-based access-control system is sound; every resource access must be via a legitimate capability containing appropriate permissions (Section 5). To formally verify the soundness, we prove our access-control system provides *capability safety* and *authority safety* properties using the *assumption-commitment* (A-C) method (Section 2.2). The capability-safety property guarantees that a process can only employ legitimately acquired capabilities that are still valid. The authority-safety property ensures that the upper bound of a process's authority is the authority of its valid capabilities, and no authority amplification occurs while delegating a capability. In addition, we prove that our design can guarantee *isolation* property based on capability-safety and authority-safety properties. The isolation property states that ac-

cessible resources by processes are mutually exclusive.

We instantiate our access-control system for MDC as the use case (**??**). Memory disaggregation in MDC makes it a suitable case for demonstrating our access-control system; a resource is a byte-addressable memory range; the access-control system must provide direct and fine-grained access to memory while hiding the physical address of objects. We describe the instantiated system, illustrate its controllers, and verify that it is sound.

Finally, we model our design and check its correctness using the *SPIN* model checker [61] (**??**). Specifically, we employ SPIN to check the safety properties based on the assumption-commitment method.

The contributions of our work are as follows.

- We present a general-purpose capability-based access-control system for disaggregated-resource architectures that covers the above qualitative criteria. Our system's design enables PEs to connect to architectures in a plug-and-play manner.
- We formally prove the soundness of our access-control design by verifying that it provides capability safety, authority safety, and isolation properties.
- We instantiate our access-control system for MDC.
- We model our access-control system and check its correctness using the *SPIN* model checker.

2 Background

This section describes the capability-object model, explain the A-C method, describe MDC, and state our threat model.

2.1 Capability-Object Model

Capability-based access-control model provides flexible techniques to enforce the least-privilege principle by sharing authority via unforgeable tokens, namely *capabilities* [37, 38]. Miller et al. [62] introduced the *capability-object model* as a security measure to control access to particular system parts, in which *object*s represent system resources and subjects. Furthermore, the model employs capabilities to encode access rights and enable objects to interact via them. A *capability* is an unforgeable token that refers to a specific object in this model.

When an object possesses a capability, it can communicate with the referred object. For example, imagine three objects: A, B, and C. Object A owns a capability referring to C and wants to delegate the capability to B. To accomplish this, A first creates two mediator objects, R and F, as depicted in Figure 1. Object A only provides B with access to F and asks objects R and F to forward B's requests; thus, B can access C through them. To revoke the access, A asks R to stop delivering B's messages, which renders B'access to F useless. Miller refers to objects R and F as *revoking facet* and *forwarding facet*, respectively. We follow Miller's approach to design our access-control system (Section 4.3).

2.2 Assumption-Commitment Method

The assumption-commitment (A-C) method [63, 64] is a compositional technique for specifying and verifying the interaction between a process and its environment, which makes it a suitable tool for reasoning about *concurrent*, *open*, and *reactive systems*. It facilitates reasoning about a system by isolating its processes and reasoning about them, where an isolated process communicates with its environment via synchronous message passing.

For reasoning about a process (p), the method requires information about four quantities: (1) The process's initial state; (2) The process inputs' properties; (3) The properties that each process's output fulfills; and (4) The process's final state. Precondition ϕ, assumption A, commitment C, and postcondition ψ provide the information for the method. Assumption A expresses what the process can assume about its prior inputs from its environment during its execution. Commitment C indicates what the environment can expect about the process's outputs. The method specifies the process with the following A-C correctness formula:

$$\langle A,\, C\rangle :\ \{\varphi\}\ p\ \{\psi\}$$

The process satisfies the A-C specification if it guarantees commitment C as long as the environment respects assumption A.

The method examines if assumption A and commitment C hold after each message exchange. It uses state machines to present and reason about the sequential segments

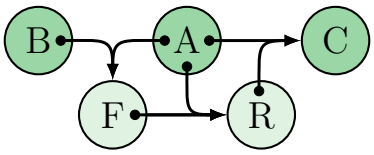


Figure 1: Capability-Object Model.
Legend: F: Forwarding Facet; R: Revoking Facet

of a system. Tuple (L, T, s, t) expresses the sequential transitional diagram of a component, in which L presents a final set of locations, T defines a final set of transitions, s is the entry point, and t is the exit location. The method refers to the above tuple as a *program*.

The A-C method leverages an *assertion network*, Q, to define the state of locations in a component's diagram. The assertion network assigns a predicate, Q_l, to each location, $l \in L$, in the diagram, which expresses the location's internal state and communication history. Furthermore, the method uses a *logical variable* to track the history of the component's communications.

The assertion network and the logical variable enable the method to reason about the component using the *assumption-commitment-inductive assertion network*, $Q(A,\ C)$, concept. To show that an assertion network is A-C-inductive, the A-C method checks if the predicate Q_l and assumption A imply predicate $Q_{l'}$ during the transition from location l to l'. Furthermore, if outgoing communication occurs during the transition, the method checks if the transition satisfies commitment C.

The method uses *Parallel Composition* rule to reason about parallel-executing processes w.r.t. their assumptions and commitments.

Rule 1. *The* A-C *correctness formula of two parallel executing processes,* $p \stackrel{def}{=} p_1 \parallel p_2$, *is as follows:*

$$\frac{\begin{array}{c}\langle A_1, C_1\rangle : \{\varphi_1\}\ p_1\ \{\psi_1\} \\ \langle A_2, C_2\rangle : \{\varphi_2\}\ p_2\ \{\psi_2\} \\ A \wedge C_1 \rightarrow A_2,\ A \wedge C_2 \rightarrow A_1\end{array}}{\langle A, C_1 \wedge C_2\rangle : \{\varphi_1 \wedge \varphi_2\}\ p\ \{\psi_1 \wedge \psi_2\}}\ \begin{array}{l}Parallel\\ Composition\end{array}$$

Where $\langle A, C_1 \wedge C_2\rangle$ are assumptions and commitments of the new process (p), regarding its environment. If A_i $(i \in \{1,\ 2\})$ contains assumptions about the connected channels to both P_1 and P_2, then C_j $(j \in \{1,\ 2\} \wedge j \neq i)$ should justify them. If A_i includes assumptions about the external channels of P_i, the new assumptions A should justify them. Verifying the conditions $A \wedge C_i \rightarrow A_j$ $(i, j \in \{1,\ 2\} \wedge i \neq j))$ leads to verifying the above conditions in p.

2.3 Memory-Driven Computing (MDC)

MDC [36] is a disaggregated-memory architecture. It realizes memory disaggregation as a shared fabric-attached memory pool. PEs and NVM modules connect via bridges

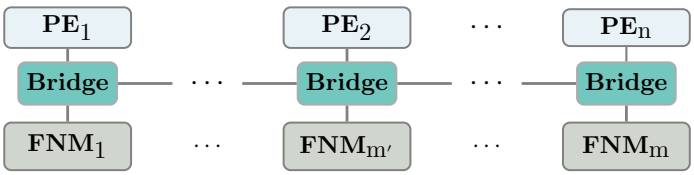


Figure 2: MDC Architecture.
Legend: FNM: Fabric-attached NVM Module

to the fast-speed fabric, as depicted in Figure 2. Near-uniform low latency of optical networking provides memory-speed access time to the NVM modules.

MDC provides distributed heterogeneous computing by enabling general-purpose CPUs and task-specific processors to attach the fabric and read/write (r/w) from/to the shared memory. Bridges route memory read/write requests to corresponding NVM modules, in which memory controllers execute them and return responses. Thus, processes can directly access all the memory addresses.

2.4 Threat Model

This work assumes a threat model where adversaries cannot physically access resource and intermediate controllers or compromise them via the network. Our system use one-way hash function and encryption to make unforgeable capabilities; thus, adversaries cannot forge new capabilities or alter existing ones. Furthermore, we assume architectures preserve the confidentiality and integrity of exchanged data by protecting connections between their elements using existing mechanisms, such as encryption. This threat model has two types of PEs: secured and unsecured. We assume an adversary can compromise any process in unsecured PEs but cannot compromise processes in secured ones. Various attacks, such as side-channel attacks, are out of the scope of this work.

3 Related Work

Following Bresniker et. al. [40] work, we reviewed 18 existing capability systems (Table 1) and compare them.

Methodology. Our methodology comprises two main steps. First, we systematically review existing literature on capability systems (i.e., Table 1), offered features by each one, their use cases, and their requirements. Then, we use the extracted features as qualitative criteria to compare existing systems against one another comprehensively. Table 1 summarizes the reviewed capability systems and the traits they cover, in which ● means that a system fully supports a trait, ◐ indicates a system partially supports a trait, and ○ denotes that a system does not satisfy a trait. The six extracted traits are as follows.

Distributed Structure. This criterion expresses that an access-control system should support a distributed system; thus, it indicates that the system should be scalable without suffering from a single point of failure problem. We refer to a capability system fully distributed if it controls access on resource sides and PE sides and can handle internal delegation locally; we indicate a system partially distributed if it only controls access on the resource sides.

Subject Granularity. This criterion refers to the granularity of subjects a system can recognize in each stage of access control. Process level is the most fine-grained subject granularity. A capability system fully supports fine-grained subject granularity if it can distinguish processes at both PE and resource sides.

Object Granularity. This criterion presents the minimum unit of a resource object that an access-control system can distinguish regarding the resource type. For example, the most fine-grained memory granularity is at the byte level.

Capability Delegation. A process should be able to delegate a subset of its capability privileges to other processes only through authorized channels.

Capability Revocation. A process should be able to revoke a delegated capability and all re-delegated capabilities in its capability-hierarchy.

Persistency. A capability system is persistence if capabilities can outlive their corresponding processes or OS reboots.

The distributed trait is a fundamental demand for disaggregated-resource architectures. Capability persistency is crucial to eliminate security risks because a persistent resource, such as persistent memory, may contain sensitive data, which remains in the resource after rebooting the system. The ability to delegate supports cooperation between processes, while the inability to revoke introduces a security vulnerability. As Table 1 depicts, each system covers 3.38 traits fully and 1.16 ones partially on average. However, nine (50%), sixteen (81%), ten (55%), and eleven (61%) of the systems cover, fully or partially, distributed, delegation, revocation, and persistency traits, respectively; none of them fully covers all the four traits. As we demonstrate in Section 4, our design fully covers all the six traits.

Now, we review capability system in more details by dividing them to Hardware-supported and OS-based systems.

Hardware-supported Systems. *CAP* [43], *StarOS* [44], *IBM System/38* [45], and CHERI [48] extend *instruction set architecture* (ISA) to protect memory in single systems. CAP proposes a hardware design cand an associated operating system to manage

Table 1: Capability-based Access-control Systems

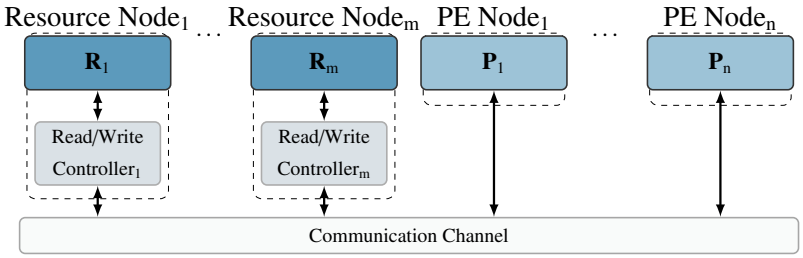
	CAP [43]	StarOS [44]	IBM/38 [45]	Hardbound [46]	iAPX432 [47]	CHERI [48]	M-Machine [39]	Hydra [49]	Chorus [50]	Amoeba [51]	Accent [52]	KeyKOS [53]	Mach [54]	EROS [55]	L4 [56]	Barrelfish [57]	CEP [58]	SemperOS [41]	SADRA
Distributed	○	●	○	○	●	○	○	○	●	●	●	○	●	○	○	●	◐	●	●
Subject Granularity	●	◐	●	●	●	●	●	●	●	●	●	●	●	●	●	●	◐	●	●
Object Granularity	●	●	●	◐	●	●	●	●	◐	◐	◐	◐	◐	◐	◐	◐	●	●	●
Capability Delegation	●	●	●	○	○	●	●	●	●	●	◐	●	●	●	●	●	●	●	●
Capability Revocation	○	○	●	○	○	○	●	○	●	◐	◐	○	◐	●	○	●	●	◐	●
Persistency	●	○	○	○	●	○	●	○	◐	●	○	◐	◐	●	●	○	●	◐	●
HW/SW Demands	ISA/OS	ISA/OS	ISA	ISO/C	ISA	ISA/C	HW/ISA	OS	OS	OS	OS	OS	OS	OS	OS	OS	Co-P	HW/OS	Co-P

Legend: ●=fully; ◐=partially; ○=no
C: Compiler; Co-P: Co-Processor; ISA: Instruction Set Architecture; OS: Operating System

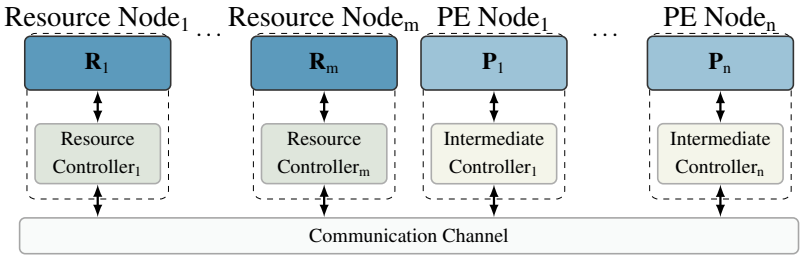
capabilities. IBM System/38 implement capabilities as pointers and protect them with tags bits. Users can pass the capabilities but cannot change their tags' value. CHERI proposed a hybrid capability model using a capability co-processor and tagged memory. Contrary to our system, these system are non-distributed ones.

StarOS is a distributed object-oriented system based on *Cm** architecture[**?**]. CEP [58] presents a distributed memory controller, implemented via co-processors.Because CEP only manages access on the memory side, it introduces communication overhead and is vulnerable to DoS attacks. Furthermore, handling local delegation/revocation operations between two processes requires communication with remote controllers. SemperOS [41] proposes a multi-kernel OS based on a hardware element, namely *data transfer unit* (DTU). It divides PEs into groups and controls each group via an independent kernel, in which kernels communicate by passing messages using DTUs. Furthermore, SemperOS assigns every PE's capabilities statically because it does not support migrating new PEs due to its capability-addressing scheme. Kernels collaborate to handle capability-related operations, which causes communication overhead and makes a system vulnerable to DoS attacks during revoking capability hierarchies. SemperOS allows spawning at most two threads per kernel to mitigate the vulnerability.

OS-based Systems. *HYDRA* [49] is an OS that treats capabilities as references to resources and allows only the kernel to manipulate them. *KeyKOS* [53], *Accent* [52], *Mach* [54], and *EROS* [55] enforce capabilities via micro-kernels. KeyKOS divides a system into domains and employs capabilities to control message passing between them. Accent proposes a distributed communication-oriented OS and provides processes with capabilities to employ communication channels. EROS provides persistent memory access at a page level. A reference monitor mediates process communication and provides transparent delegation/revocation via forwarding objects. Mach [54] is an object-oriented OS that enables processes to pass data and capabilities via a messaging system.



(a) Abstract Disaggregated System.



(b) Distributed Access-control System.

Figure 3: Distributed Systems.

4 Proposed Access-Control System

We introduce the design of our capability-based access-control system in this section. We first present an abstract model for disaggregated systems; then, we introduce *access controller*s and integrate them into this model; finally, we demonstrate our capability model.

4.1 Abstract Disaggregated System

We construct an abstract disaggregated-system model to focus on the general characteristics of disaggregated systems. The model comprises *resource nodes* ($\mathbb{N}_r$), *PE nodes* ($\mathbb{N}_p$), resources ($\mathbb{R}$), *processes* ($\mathbb{P}$), and a *communication medium*, as illustrated in Figure 3a. Each resource node $N_r^j \in \mathbb{N}_r$ ($1 \leqslant j \leqslant m$) contains a subset of resources $R_j \subseteq \mathbb{R}$, and each PE node $N_p^k \in \mathbb{N}_p$ ($1 \leqslant k \leqslant n$) hosts a subset of processes $P_k \subseteq \mathbb{P}$. A resource node operates autonomously from other resource nodes. It contains a controller which handles r/w requests. Processes and controllers communicate using the request/response pattern, exchanging messages via the medium. We designed our access-control system based on this model.

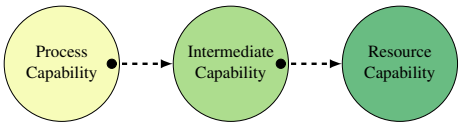


Figure 4: The Capability Model.

4.2 Access Controllers

Imagine a resource node as a country's capital and all the PE nodes as provinces. The central governor delegates a subset of its power to local governors. It only controls access to the capital's resources or handles the relationship between two citizens from different provinces. Hence, the model distributes the central government's workload and reduces the exchanged messages between central and provincial governors. We followed such a model to design a scalable distributed access-control system.

To adapt the model, we designed two controller types: *resource* and *intermediate controllers*, as illustrated in Figure 3b. Resource controller $RC_j \in \mathbb{RC}$ $(1 \leqslant j \leqslant m)$ acts as the central governor in resource node N_r^j, and each intermediate controller $IC_k \in \mathbb{IC}$ $(1 \leqslant k \leqslant n)$ serves as the local governor in PE node N_p^k. Neither resource nor intermediate controllers interact with the controllers of their types. In addition, processes cannot directly communicate with resource nodes or share their access rights with others.

Intermediate controllers handle two main tasks. First, they perform the first stage of access control by inspecting requests and checking if the process possesses the appropriate access rights, regarding their requests. They deny requests if checking fails. Second, they handle local delegation/revocation requests. For example, assume two processes, p, $p' \in P_k$, where p can access resource $r \in R_j$ and wants to share this privilege with p'. Controller IC_k performs the delegation without involving RC_j.

Resource controllers fulfill three main tasks: *allocating resources*, *delegating/revoking access rights*, and *enforcing access rights*. They assign each resource in their nodes to a specific process by performing the first task. Furthermore, they enable access-right delegation/revocation when delegator and receiver processes run in different PE nodes. For example, assume $p \in P_k$ requests to share its access rights to resource $r \in R_j$ with $p' \in P_{k'}$. Resource controller RC_j handles the request, cooperating with intermediate controllers IC_k and $IC_{k'}$. Moreover, they perform the second stage of access control. In the previous example, RC_j declines p''s requests regarding resource r if process p has revoked the permission. Controllers employ the capability tokens to execute their tasks.

4.3 Capability Model

We desire our capability model to reflect the delegated control mechanism between resource and intermediate controllers. Intermediate controllers prevent processes from contacting resource controllers directly; thus, resource controllers only accept requests from intermediate controllers. Now that we have designed our access controllers, we introduce our capability-object model

Our capability model comprises three capability types: *resource* ($\mathbb{C}_r$), *intermediate*

($\mathbb{C}_i$), and *process* ($\mathbb{C}_p$) capabilities, as depicted in Figure 4. As the naming suggests, each type belongs to its corresponding controller/process in our design and contains information about the access rights of a specific process to a particular resource. In addition, process and intermediate capabilities include a pointer, pointing to their corresponding intermediate and resource capabilities. We use operator $\mapsto$ to represent the point-to relation between two capabilities. For example, assume that capabilities $c_p \in \mathbb{C}_p$, $c_i \in \mathbb{C}_i$, and $c_r \in \mathbb{C}_r$, where c_p and c_i point to c_i ($c_p \mapsto c_i$) and c_r ($c_i \mapsto c_r$), respectively. These three capabilities contain the same permissions to access resource $r \in R_j$.

Indicator Flag. A process must indicate the exact delegated capability when it intends to revoke the capability. Thus, our access-control system requires a mechanism to track delegated capabilities; it provides the mechanism using a boolean flag, namely *indicator*, inside capabilities. When the flag is set, the containing capability points to a delegated capability. We refer to a capability when its indicator flag is set simply as an **indicator** and clarify how our system leverages it when explaining capability-related operations (Section 4.3.2). It is worth noting that processes and controllers cannot employ indicators for read/write operations.

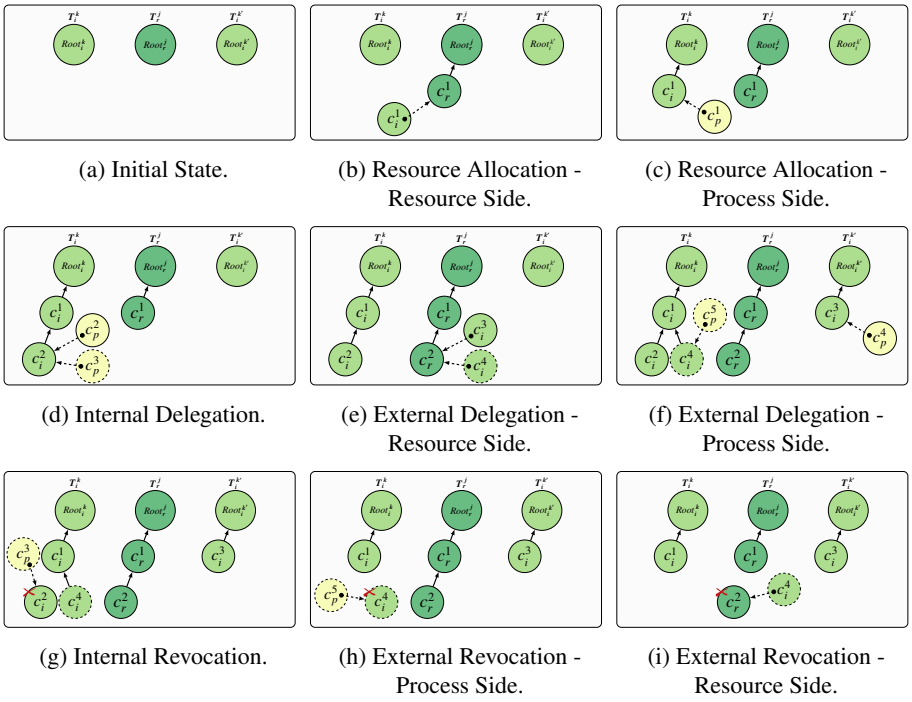
4.3.1 Capability Tree

Controllers preserve their capabilities inside *rooted tree*s [**?**]. A rooted tree is a tree with a particular vertex as its *root*. We refer to the trees as *capability tree*s ($\mathbb{T}_r \cup \mathbb{T}_i$), where the root capabilities belong to controllers. A subtree in a capability tree represents a *delegation hierarchy* in which a child node indicates a delegation from its parent capability. A controller only possesses the authority to add/remove capabilities to/from its tree and leverages it to handle capability-related operations. Controllers retain their trees in Non-Volatile memories. It enables controllers to support capability persistency.

4.3.2 Capability-related Operations

Capability-related operations include allocating resources and delegating/revoking capabilities. We demonstrate how controllers leverage capability trees to accomplish their tasks through an example. Imagine a disaggregated system where process $p \in P_k$ requests gaining *read/write* (r/w) access to resource $r \in R_j$ (*resource allocation*). After gaining access, p delegates the received privileges to process $p' \in P_k$ in the same PE node (*internal delegation*) and process $p'' \in P_{k'}$ in another PE node (*external delegation*). Finally, process p revokes both delegated capabilities (*internal* and *external revocation*). **??** 4.3.1–5i illustrate how the controllers modify their capability trees to accomplish their tasks in the above scenario. The controllers' trees contain only their root capabilities at the initial state, as depicted in Section 4.3.1.

Resource Allocation. By receiving the request, resource controller RC_j allocates the resource and creates a resource capability, c_r^1, which comprises data about p and r. Then, it inserts the capability as the root's child inside its tree and forges the corresponding intermediate capability, c_i^1, as illustrated in Figure 5b. To complete its task, controller RC_j sends c_i^1 to intermediate controller IC_k.



(a) Initial State. (b) Resource Allocation - Resource Side. (c) Resource Allocation - Process Side.

(d) Internal Delegation. (e) External Delegation - Resource Side. (f) External Delegation - Process Side.

(g) Internal Revocation. (h) External Revocation - Process Side. (i) External Revocation - Resource Side.

Figure 5: Capability Trees.
Legend: ○ = process capability; ○ = intermediate capability; ● = resource capability
○ = indicator flag is unset; ◌ = indicator flag is set;
●- - -►: pointed-to connection in our capability-object model; ⟶: child-parent relation in a capability tree

The Intermediate Controller adds the received capability to its capability tree (Figure 5c). Furthermore, IC_k forges the corresponding process capability, c_p^1, and sends it to process p. Hence, p possesses an unforgeable process capability.

Internal Delegation. Control IC_k handles delegating the capability to p' because both processes run on the same node. The controller generates intermediate capability c_i^2 and inserts it as the child of capability c_i^1 by receiving the request (Figure 5d). Furthermore, the controller forges the corresponding process capability, c_p^2, and sends it to p'. Process p' can now employ the received capability to execute r/w operations over resource r. Moreover, controller IC_k forges indicator c_p^3 for p. Process p can employ the indicator to revoke the delegated capability; however, it can neither apply the indicator during r/w requests nor re-delegate it.

External Delegation. Intermediate controller IC_k collaborates with resource controller RC_j to handle the external delegation. Controller RC_j generates and inserts resource capability c_r^2 to its tree due to the request (Figure 5e). Furthermore, RC_j forges intermediate capability c_i^3 for controller $IC_{k'}$ and indicator c_i^4 for controller IC_k.

Both intermediate controllers add the received capabilities to their trees (Figure 5f). Controller $IC_{k'}$ creates process capability c_p^4 for process p'', and controller IC_k forges indicator c_p^5 for process p. At this stage, processes p, p', and p'' possess the capability sets $\{c_p^1,\ c_p^3,\ c_p^5\}$, $\{c_p^2\}$, and $\{c_p^4\}$, respectively.

Internal Revocation. Process p leverages indicator c_p^3 to revoke the delegated capability to process p'. When intermediate controller IC_k receives the request, it realizes that the indicator points to intermediate capability c_i^2 inside its tree (Figure 5g). Hence, the controller performs an internal revocation only by removing capability c_i^2 from its tree. Process p' cannot use process capability c_p^3 any further because the process capability points to a non-existing intermediate capability in the capability tree.

External Revocation. Process p employs indicator c_p^5 to revoke the delegated capability to p''; by receiving the revocation request, controller IC_k notices that the indicator points to another indicator, c_i^4, in its tree. Hence, the controller removes indicator c_i^4 from its tree at the first phase (Figure 5h). Afterward, the controller forwards the request along with indicator c_i^4 toward resource controller RC_j.

When controller RC_j receives the request, it notices that indicator c_i^4 points to capability c_r^2 inside its capability tree. The controller removes capability c_r^2 from its tree (Figure 5i), which completes the external revocation.

At this stage, the p'''s request, comprising process capability c_p^4, to read/write from/to resource r passes the first access check at controller $IC_{k'}$ because c_p^4 points to an existing intermediate capability, c_i^3, inside the controller's tree. Thus, controller $IC_{k'}$ forwards the request along with c_i^3 to resource controller RC_j. However, controller RC_j returns a failure response to $IC_{k'}$ because c_i^3 points to capability c_r^2, which no longer exists in the controller's tree. Consequently, intermediate controller $IC_{k'}$ removes capability c_i^3 from its tree and forwards the failure response to process p''.

5 System Design's Soundness

Now that we have seen how our system controls access, we prove that our capability-based access control system is sound. Following Maffeis et al. [**?**] work, we prove our system design's soundness by demonstrating that it satisfies the capability-safety and authority-safety properties. Furthermore, we prove that our system guarantees isolation property based on these two properties.

In the rest of the section, we employ a few auxiliary functions in the definitions and proofs. Table 2 lists the functions and their descriptions, in which $\mathbb{V} = \{r, w\}$ is the permission set (r: read, w: write), and $\mathbb{R}$ denotes the set of all resources.

5.1 Well-Formed Capability Trees

To define well-formed capability trees, we must first specify the partial order.

Partial Order. We define the partial order ($\prec$) between two capabilities ($c' \prec c$) as follows:

$$c' \prec c \iff rsrc(c') \subseteq rsrc(c) \ \wedge \ priv(c') \subseteq priv(c)$$

We use the partial-order property to specify the relation between capabilities in a delegation hierarchy.

Resource-capability Tree. A resource-capability tree $T_r \in \mathbb{T}_r$ is a rooted tree with the root $c_r^{root} \in \mathbb{C}_r$. The root capability belongs to the resource controller, points to all the node's resources, and possesses all the permissions. The controller generates no corresponding intermediate capability for it. Tree T_r is a well-formed resource-capability tree if all its sub-trees are partially-ordered delegation hierarchy as follows:

$$\forall c_r,\ c'_r \in \mathbb{C}_r : isInTree(T_r,\ c_r,\ c'_r) \Rightarrow c'_r \prec c_r$$

Theorem 5.1. *A well-formed resource tree $T_r \in \mathbb{T}_r$ remains well-formed after performing capability-related operations.*

Proof. The proof proceeds by structural induction on T_r and comprises a base case and three inductive cases.

- **Base Case:** T_r only contains the root capability.
  In this case, the tree satisfies the well-formness' condition.
- **Inductive Case One:** T_r is a well-formed capability tree, and the resource controller (*RC*) allocates a resource.
  After allocating the resource, *RC* forges the corresponding capability (c_r) for it such that $c_r \prec c_r^{root}$; then, *RC* inserts it to T_r as the child of the root. Hence, the resource-allocating exercise retains the tree's well-formedness.
- **Inductive Case Two:** T_r is a well-formed capability tree, and *RC* performs an external capability-delegation operation over c_r.
  RC delegates c_r by constructing c'_r such that $c'_r \prec c_r$. Afterward, *RC* adds c'_r into

T_r as the child of c_r. Thus, the external-delegating operation preserves the tree's well-formedness.

- **Inductive Case Three:** T_r is a well-formed capability tree, and *RC* performs an external capability-revocation operation over c_r.
  RC removes c_r and all the capabilities in its subtree. Thus, the external-revocation operation preserves the tree's well-formedness.

□

Intermediate-capability Tree. An intermediate-capability tree $T_i \in \mathbb{T}_i$ is a rooted tree with the root $c_i^{root} \in \mathbb{C}_i$. The root capability belongs to the intermediate controller, but it points to no resource and possesses no permissions; thus, the controller generates no corresponding process capability for it. Tree T_i is a well-formed intermediate-capability tree if all its sub-trees (not including the root) are partially-ordered delegation hierarchy as follows:

$$\forall\, c_i, c_i' \in \mathbb{C}_i : \mathit{isInTree}(T_i,\, c_i,\, c_i') \,\wedge\, \neg \mathit{isRoot}(T_i,\, c_i) \Rightarrow c_i' \prec c_i$$

Theorem 5.2. *A well-formed intermediate tree $T_i \in \mathbb{T}_i$ remains well-formed after performing capability-related operations.*

Proof. The proof proceeds by structural induction on T_i and comprises a base case and three inductive cases.

- **Base Case:** T_i only contains the root capability.
  In this case, the tree satisfies the well-formness' condition.
- **Inductive Case One:** T_i is a well-formed capability tree, and the resource controller (*IC*) receives an intermediate capability or an indicator due to a resource allocation or an external delegation, respectively. *IC* inserts the received capability or indicator to T_i as the child of the root. Hence, allocating resource or externally delegating a capability retain the tree's well-formedness.
- **Inductive Case Two:** T_i is a well-formed capability tree, and *IC* performs an internal capability-delegation operation over c_i.
  IC delegates c_i by constructing c_i' such that $c_i' \prec c_i$. Afterward, *IC* adds c_i' into T_i as the child of c_i. Thus, the internal-delegating operation preserves the tree's well-formedness.
- **Inductive Case Three:** T_i is a well-formed capability tree, and *IC* performs an internal capability-revocation operation over c_i.
  IC only removes c_i and all the capabilities in its subtree. Thus, the internal-revocation operation preserves the tree's well-formedness.

□

Table 2: Function Definitions

Function Name	Function Signature	Description
rsrc	$\mathbb{C} \rightarrow \mathbb{R}$	returns the pointed resources inside a capability.
priv	$\mathbb{C} \rightarrow 2^{\mathbb{V}}$	returns the permissions inside a capability
getRoot	$\mathbb{T}_x \rightarrow \mathbb{C}_x$	returns the root of a capability tree.
isRoot	$\mathbb{T}_x \times \mathbb{C}_x \rightarrow boolean$	checks if a capability is the root of a capability tree
isInTree	$\mathbb{T}_x \times \mathbb{C}_x \times \mathbb{C}_x \rightarrow boolean$	checks if the second capability is inside a tree, where the first capability indicates the search's start point.

Legend: $x \in \{r, i\}$, $\mathbb{C} = \{\mathbb{C}_r \cup \mathbb{C}_i \cup \mathbb{C}_p\}$

5.2 Valid Capability

A capability is valid if the capability owner can employ it to execute read/write operations over the pointed resource by the capability. Because an intermediate controller or a process can send a request, we demonstrate the validity of a capability at both levels.

5.2.1 Valid Intermediate Capability

An intermediate capability is valid if a resource controller has forged it and it points to an existing resource capability inside the controllers tree. For example, consider capabilities c_i^1 and c_i^3 in Figure 5i. c_i^1 points to an existing capability (c_r^1) inside T_r^j. Hence, IC_k can employ it to exercise the read/write operations, which makes c_i^1 a valid capability. At the same time, c_i^3 is an invalid capability because it points to a removed resource capability (c_r^2) from T_r^j. We formally define a valid intermediate capability as follows:

$$
\begin{aligned}
isValidIntCap(c_i) \stackrel{\text{def}}{=}\ & \exists\, c_r \in \mathbb{C}_r,\ T_r \in \mathbb{T}_r : \\
& c_i \mapsto c_r \wedge \\
& isInTree(T_r,\ getRoot(T_r),\ c_r)
\end{aligned}
$$

Resource controllers can only forge valid intermediate capabilities or invalidate them by removing the pointed resource capabilities from their trees.

5.2.2 Valid Process Capability

A process capability is valid if it points to an existing resource capability inside a resource controller's tree. In addition, the pointed intermediate capability must be valid or be in the delegation hierarchy of a valid one. We denote the capability on top of a delegation hierarchy as an *original* intermediate capability. It means a resource controller has forged the capability with an unset indicator flag. Let $(S^i,\ c_i)$ be a rooted subtree of (T_i, c_i^{root}), in which node c_i denotes the subtree's root and is the child of c_i^{root}. Subtree S^i indicates a delegation hierarchy inside T_i. Capability c_i is original, while the rest of the capabilities in S^i are not. For example, consider capabilities c_i^1 and c_i^2 inside tree T_i^k in Figure 5d. c_i^1 is an original capability while c_i^2 is not.

Resource controllers only accept original intermediate capabilities for read/write operations because they have forged them and can check their integrity. In the above

example, suppose process p employs c_p^2 to read from resource r. Although c_p^2 points to c_i^2, IC_k cannot replace c_p^2 with c_i^2 when it forwards the request to RC_j. Instead, IC_k first replaces c_p^2 with c_i^1 as it is the original capability of c_i^2; then, IC_k forwards the request.

We define function **getOriginalCap**: $\mathbb{T}_i \times \mathbb{C}_i \rightarrow \mathbb{C}_i$ to find an original capability inside an intermediate-capability tree. The function traverses the tree from a given capability toward the root until it reaches an original capability in the path. We formally define a valid process capability as follows:

$$
\begin{aligned}
isValidProCap(c_p) \stackrel{\text{def}}{=} \exists\, c_i,\, c_i' \in \mathbb{C}_i,\, T_i \in \mathbb{T}_i :\\
c_p \mapsto c_i \wedge\\
isInTree(T_i,\, getRoot(T_i),\, c_i) \wedge\\
c_i' = getOriginalCap(T_i,\, c_i) \wedge\\
isValidIntCap(c_i')
\end{aligned}
$$

For instance, in Figure 5h, c_p^4 is a valid capability while c_p^5 is not because IC_k has removed c_i^4 from its capability tree. Now that we have expressed the well-formed capability trees and specified valid capability, we define the security properties.

5.3 Security Properties

Our access-control system guarantees three security properties, including: *capability safety*, *authority safety*, and *isolation*. We prove our system satisfies these properties by demonstrating that our design handles capability-related operations correctly; thus, we first define five lemmas, one for each operation, and verify them using the assumption-commitment technique.

Our modeling has three component types: *process*, *resource controller*, and *intermediate controller*. In this modeling, the components interact to execute an operation. We consider five operations in this modeling: resource allocation, local capability delegation, remote capability delegation, local capability revocation, and remote capability revocation. In addition, we define five lemmas based on the operations and explain the operations' precondition, postcondition, assumption, and commitment properties to prove the lemmas based on the assumption-commitment (*A*-*C*) technique. Each proof considers the states of the components involved in the corresponding operation.

5.4 First Lemma: Resource Allocation

The first lemma indicates that any resource-allocation operation creates a valid p-cap. We formally define the lemma and proof it.

Lemma 1. *Given a process with a unique PID in a process node with a unique NID, an intermediate controller in the same node, and an resource controller in a resource node with a unique NID, the process will get a valid p-cap by sending a resource-allocation request.*

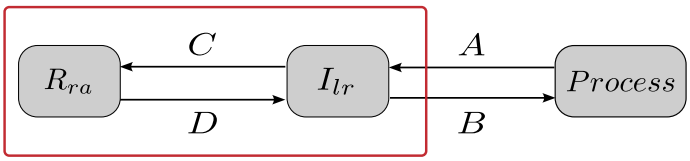


Figure 6: Syntactic Interfaces of the Resource-allocation Programs.

Proof. We use the assumption-commitment technique to prove the lemma.

To prove the lemma, we model the intermediate and the resource controllers in a resource-allocation operation by programs $I_{ra} \stackrel{\text{def}}{=} (L^i_{ra}, T^i_{ra}, s^i_{ra}, t^i_{ra})$ and $R_{ra} \stackrel{\text{def}}{=} (L^r_{ra}, T^r_{ra}, s^r_{ra}, t^r_{ra})$, respectively. The programs and the process exchange messages via synchronous channels to handle resource-allocation requests. Figure 6 depicts these two programs and their syntactic interfaces. In addition, figures 7a and 7b illustrate sequential diagrams of programs I_{ra} and R_{ra}, respectively. In the next step, we prove that when the process sends a resource-allocation request, it will receive a valid p-cap.

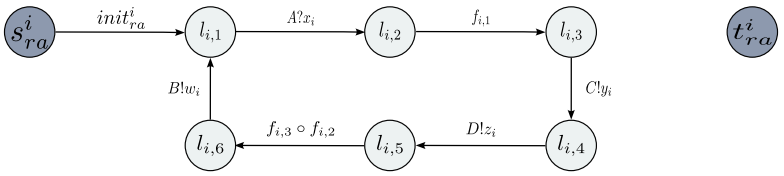
5.4.1 Resource Allocation - Intermediate Controller

Program I_{ra} receives a resource-allocation request from the process via channel A, creates a corresponding request for program R_{ra}, and sends it through channel C. Furthermore, It receives the response from program R_{ra} via channel D. Program I_{ra} updates its capability tree when it receives an i-cap from the resource controller by inserting the capability inside the tree. In addition, it creates the corresponding p-cap for the inserted i-cap. Finally, it creates a response for the process and sends the p-cap inside the response to the process through channel B. The functions of I_{ra} are as follows:

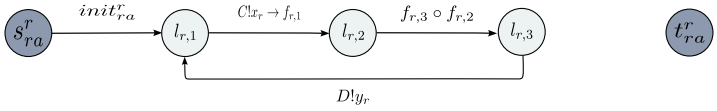
$$
\begin{aligned}
init^i_{ra}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(T_i))),\\
f_{i,1}(\sigma) &= (\sigma : y_i \mapsto genRaReq(\sigma(x_i))),\\
f_{i,2}(\sigma) &= (\sigma : pCap \mapsto insert(\sigma(rootPtr), \sigma(z_i.cap)))\\
f_{i,3}(\sigma) &= (\sigma : w_i \mapsto genRaRes(\sigma(x_i), \sigma(pCap)))
\end{aligned}
$$

The A-C formula for program I_{ra} is as follows:

$$\vdash \langle Ass^i_{ra},\ C^i_{ra}\rangle\ \{\varphi^i_{ra}\}\ I_{ra}\ \{\psi^i_{ra}\}$$



(a) Program I_{ra}.



(b) Program R_{ra}.

Figure 7: Resource-allocation Programs.

where the formal definition of φ^i_{ra}, ψ^i_{ra}, Ass^i_{ra}, and C^i_{ra} are as follows:

$$
\begin{array}{lll}
\varphi^i_{ra} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| \geqslant 1 \\
\psi^i_{ra} & \stackrel{\text{def}}{=} & false \\
Ass^i_{ra} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & \quad isRaReq(last(A)) \ \wedge \ isUniquePID(last(A).spid)) \wedge \\
 & & (\#C = \#D > 0 \rightarrow \\
 & & \quad isRaRes(last(D)) \ \wedge \ isValidCap(last(D).cap)) \\
C^i_{ra} & \stackrel{\text{def}}{=} & (\#A = \#B = \#C = \#D > 0 \rightarrow \\
 & & \quad isRaRes(last(B)) \ \wedge \ isValidCap(last(B).cap))) \wedge \\
 & & (\#C > 0 \rightarrow \\
 & & \quad isRaReq(last(C)) \ \wedge \ isUniquePID(last(C).spid)))
\end{array}
$$

Wherein $icTree$ is the capability tree of the intermediate controller.

The assertion network for I_{ra} is as follows:

$$Q_{s^i_{ra}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |T_i| \geqslant 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isRaReq(last(A)) \ \wedge \ isUniquePID(last(A).spid)$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isRaReq(y_i) \ \wedge \ isUniquePID(y_i.spid)$$

$$Q_{l_{i,4}} \stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \ \wedge \ last(D) = z_i \ \wedge \ isRaRes(last(D)) \ \wedge \ isValidCap(last(D).cap)$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \ \wedge \ last(D) = z_i \ \wedge \ isRaRes(w_i) \ \wedge \ isValidCap(w_i.cap)$$

$$Q_{t^i_{ra}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^i_{ra}, commitment C^i_{ra}, and channels A, B, C, and D.

- $\models Q_{s^i_{ra}} \rightarrow C^i_{ra}$ follows from the above definitions.
- $\models Q_{s^i_{ra}} \wedge A^i_{ra} \rightarrow Q_{l_{i,1}} \circ init^i_{ra}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ A^i_{ra} \ \rightarrow \ ((A^i_{ra} \ \rightarrow \ Q_{l_{i,2}}) \ \wedge \ C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Because $\#A > 0$, the first implication of Ass_i satisfies $isRaReq(last(A))$ and $isUniquePID(last(A).spid)$. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ A^i_{ra} \ \rightarrow \ Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ creates a resource-allocation request for program R_{rr} and assigns it to variable y_i, such that $isRaReq(y_i) = true$ and $y_i.spid = last(A).spid$. In addition, $\sigma' \models isUniquePID(y_i.spid)$ because , from A^i_{ra}, we have $isUniquePID(last(A).spid) = true$. Thus, this verification holds.
- $\models Q_{l_{i,3}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,4}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{ra} sends y_i through channel C. C^i_{ra} holds because $last(C) = \sigma(y_i)$ and $\sigma \models isRaReq(y_i) \wedge isUniquePID(y_i.spid)$. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,5}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i, h \mapsto v, \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D just took place, and function g assigns the received value to z_i. Since $\#D > 0$, from A^i_{ra} we have $\sigma' \models isRaRes(z_i) \ \wedge \ isValidCap(z_i.cap)$ Thus, this verification holds.
- $\models Q_{l_{i,5}} \wedge A^i_{ra} \rightarrow Q_{l_{i,6}} \circ (f_{i,3} \circ f_{i,2})$. In this internal transition, function $f_{i,2}$ inserts the intermediate capability $iCap$ into T_i and returns $pCap$ that points to $iCap$ inside the tree. Hence, $\sigma' \models |T_i| > 1$. Because $\sigma \models isValidCap(iCap)$ and $pCap$ points to the inserted $iCap$ inside T_i, we have $\sigma' \models isValidCap(pCap)$. Function $f_{i,3}$ creates a response for the process and assigns it to w_i, wherein $w_i.cap = pCap$. Consequently, $\sigma' \models isRaRes(w_i) \ \wedge \ isValidCap(w_i.cap)$. Thus, this verification holds.
- $\models Q_{l_{i,6}} \wedge A^i_{ra} \rightarrow ((A^i_{ra} \rightarrow Q_{l_{i,1}}) \wedge C^i_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{ra} sends w_i through channel B. C^i_{ra} holds because $\sigma \models isRaRes(w_i) \ \wedge \ isValidCap(w_i.cap)$. Hence, this verification holds.

Since $Q_{t^i_{ra}} \rightarrow \psi^i_{ra}$, the post-condition of program I_{ra} is satisfied.

5.4.2 Resource Allocation - Resource Controller

Program R_{ra} receives a request for allocating a resource from program I_{ra} via channel C. In the next step, program R_{ra} allocates the requested resource and inserts its corresponding resource capability inside its capability tree. Furthermore, it creates the corresponding intermediate capability of the resource capability. Finally, program R_{ra} creates a response, inserts the intermediate capability, and sends the response message through channel D toward program I_{ra}. The conditions and functions of R_{ra} are as follows:

$$
\begin{aligned}
init^r_{ra}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(T_r))),\\
f_{r,1}(\sigma) &= (\sigma : rCap \mapsto allocRes(\sigma(x_r))),\\
f_{r,2}(\sigma) &= (\sigma : iCap \mapsto insert(\sigma(T_r), \sigma(rootPtr),\\
&\qquad \sigma(rCap))),\\
f_{r,3}(\sigma) &= (\sigma : y_r \mapsto genRaRes(\sigma(x_r), iCap))
\end{aligned}
$$

where T_r is the resource-capability tree, $rCap$ is a resource capability , and $iCap$ is an intermediate capability.

The A-C formula for process R_{ra} is as follows:

$$\vdash \langle A^r_{ra} \ C^r_{ra} \rangle \ \{\varphi^r_{ra}\} \ R_{ra} \ \{\psi^r_{ra}\}$$

where the formal definition of φ_r, ψ_r , A^r_{ra}, and C^r_{ra} are as follows:

$$\varphi^r_{ra} \stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge \ |T_r| = 1$$

$$\psi^r_{ra} \stackrel{\text{def}}{=} \mathit{false}$$

$$A^r_{ra} \stackrel{\text{def}}{=} \#C > 0 \rightarrow (\mathit{isRaReq}(\mathit{last}(C)) \ \wedge \ \mathit{isUniquePID}(\mathit{last}(C).\mathit{spid}))$$

$$C^r_{ra} \stackrel{\text{def}}{=} (\#C = \#D > 0) \rightarrow (\mathit{isRaRes}(\mathit{last}(D)) \ \wedge \ \mathit{isValidCap}(\mathit{last}(D).\mathit{cap}))$$

The assertion network for R_{ra} is as follows:

$$Q_{s^r_{ra}} \stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge \ |T_r| = 1$$

$$Q_{r_1} \stackrel{\text{def}}{=} \#C = \#D$$

$$Q_{r_2} \stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ \mathit{last}(C) = x_r \ \wedge \ \mathit{isRaReq}(\mathit{last}(C)) \wedge \mathit{isUniquePID}(\mathit{last}(C).\mathit{spid}) \ \wedge \ \mathit{isValidCap}(\mathit{rCap})$$

$$Q_{r_3} \stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ \mathit{last}(C) = x_r \wedge \mathit{isRaRes}(y_r) \ \wedge \ \mathit{isValidCap}(y_r.\mathit{cap})$$

$$Q_{t^r_{ra}} \stackrel{\text{def}}{=} \mathit{false}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^r_{ra}, commitment C^r_{ra}, and channels C and D.

- $\models Q_{s^r_{ra}} \rightarrow C^r_{ra}$ follows from the above definitions.
- $\models Q_{s^r_{ra}} \ \wedge \ A^r_{ra} \rightarrow Q_{l_{r,1}} \circ \mathit{init}^r_{ra}$. In this internal transition, the size of the capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{r,1}} \ \wedge \ A^r_{ra} \ \rightarrow \ ((A^r_{ra} \ \rightarrow \ Q_{l_{r,2}}) \ \wedge \ C^r_{ra}) \circ (f_{r,1} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r, \ h \mapsto v, \ \sigma(h).(C, v))$ for arbitrary v. In this input transition, a communication through channel C just took place. Function g assigns the last received value from channel C to x_r. Since $\#C > 0$, the first implication of A^r_{ra} satisfies $\mathit{isRaReq}(\mathit{last}(C))$ and $\mathit{isUniquePID}(\mathit{last}(C).\mathit{spid})$ predicates. Function $f_{r,1}$ allocates the requested resource and assigns the returned valid r-cap to rCap. Hence, Q_{r_2} satisfies $\mathit{isValidCap}(\mathit{rCap})$ predicate. C^r_{ra} holds since $\#C \neq \#D$. Thus, this verification holds.
- $\models Q_{l_{r,2}} \ \wedge \ A^r_{ra} \rightarrow Q_{l_{r,3}} \circ (f_{r,3} \circ f_{r,2})$. In this internal transition, function $f_{i,2}$ inserts rCap into T_r. Hence, $\sigma' \models |T_r| > 1$. Furthermore, function $f_{i,2}$ returns a valid i-cap, which points to the inserted resource capability inside the tree. Variable iCap keeps the returned intermediate capability. Because iCap points to the inserted rCap inside T_r and $\sigma \models \mathit{isValidCap}(\mathit{rCap})$, we have $\sigma' \models \mathit{isValidCap}(\mathit{iCap})$. Function $f_{i,3}$ creates a resource-allocation response for program I_{ra} and assigns it to y_r, such that $\mathit{isRaRes}(y_r) = \mathit{true}$ and $y_r.\mathit{cap} = \mathit{iCap}$. In addition, because $\mathit{isValidCap}(\mathit{iCap})$, we have $\mathit{isValidCap}(y_r.\mathit{cap})$. Thus, this verification holds.

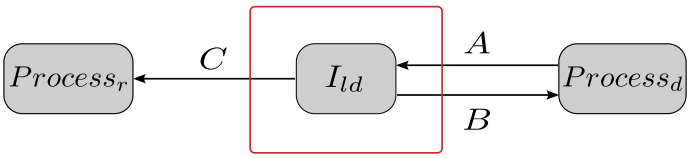


Figure 8: Syntactic Interfaces of the Local-delegation Programs.

- $\models Q_{l_{r,3}} \wedge A^r_{ra} \rightarrow ((A^r_{ra} \rightarrow Q_{l_{r,1}}) \wedge C^r_{ra}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_r)))$. In this output transition, program R_{ra} just sends y_r through channel C toward program I_{ra}. C^r_{ra} holds because $\#C = \#D > 0$, $last(D) = \sigma(y_r)$, and $\sigma \models isRaRes(y_r) \wedge isValidCap(y_i.cap)$. Hence, this verification holds.

Since $Q_{t^r_{ra}} \rightarrow \psi^r_{ra}$, the post-condition of program R_{ra} is satisfied.

5.4.3 Resource Allocation - Parallel Composition

This section presents the parallel composition of the resource-allocation components based on the described rule in Section 2.2. Consider the parallel composition $R_{ra} \parallel I_{ra}$. By applying the parallel composition rule, we deduce the following A-C formula:

$$\begin{gathered} \vdash \langle A_{ra},\ C_{ra} \rangle \\ \{\#A = \#B = \#C = \#D = 0 \ \wedge\ |T_r| = |T_i| = 1\} \\ R_{ra} \parallel I_{ra}\ \{false\} \end{gathered}$$

where A_{ra} and C_{ra} are as follows:

$$\begin{aligned} A_{ra} &\stackrel{\text{def}}{=} \#A > 0 \rightarrow \\ &\qquad isRaReq(last(A)) \ \wedge\ isUniquePID(last(A).spid)) \\ C_{ra} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D > 0 \rightarrow \\ &\qquad (isRaRes(last(B)) \ \wedge\ isValidCap(last(B).cap)) \end{aligned}$$

Because $\models A_{ra} \ \wedge\ C^i_{ra} \rightarrow A^r_{ra}$ and $\models A_{ra} \ \wedge\ C^r_{ra} \rightarrow A^i_{ra}$. Thus, $R_{ra} \parallel I_{ra}$ generates a valid capability when it receives a resource-allocation request from a process.

□

5.5 Second Lemma: Internal Delegation

The second lemma indicates that the local delegation of a valid p-cap by a process creates a valid p-cap for another process inside the same node. Hence, the intermediate controller handles the local delegation inside the node. We formally define the lemma and proof it.

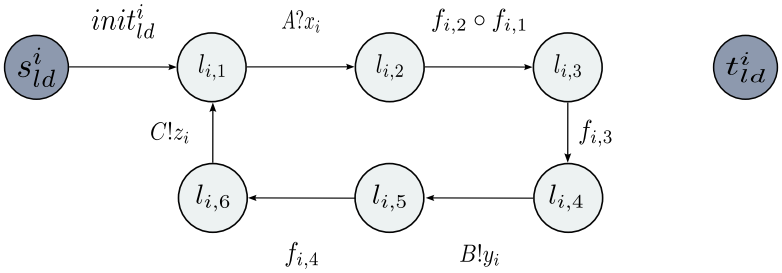


Figure 9: Internal Delegation - Program I_{ld}.

Lemma 2. *Given two processes with unique IDs inside the same process node, delegation of a valid capability from the first process, $process_d$, to the second process, $process_r$, causes $process_r$ and $process_d$ to receive a valid process capability and a valid indicator capability, respectively.*

Proof. We use the assumption-commitment technique to prove the lemma.

We model the intermediate controller in a local-delegation operation by program $I_{ld} \stackrel{\text{def}}{=} (L_{i_ld}, T_{i_ld}, s_{i_ld}, t_{i_ld})$ to prove the lemma. Figure 8 depicts the syntactic interfaces between the program and two processes. In addition, figure 9 illustrates sequential diagrams of programs I_{ld}. In the next step, we prove that $process_r$ receives a valid p-cap when $process_d$ delegates a valid capability. In addition, we prove that $process_d$ receives a valid p-cap of type indicator to the new capability inside the intermediate-capability tree.

5.5.1 Local Delegation - Intermediate Controller

Program I_{ld} receives a local capability-delegation request from $process_d$ via channel A. The request contains a valid p-cap that points to an intermediate capability inside the intermediate-capability tree. In the next step, program I_{ld} delegates the intermediate capability according to the request and inserts the new intermediate capability inside the tree. Finally, program I_{ld} creates a delegation response for $procces_r$ and a delegation-confirmation response for $process_d$, respectively. The delegation response and the delegation-confirmation response contain the new p-cap and the new indicator capability, respectively. The functions of I_{ld} are as follows:

$$
\begin{aligned}
init^i_{ld}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : indCap \mapsto delegate(\sigma(rootPtr), \sigma(x_i))),\\
f_{i,2}(\sigma) &= (\sigma : pCap \mapsto genCapFromInd(\sigma(icTree), \sigma(indCap))),\\
f_{i,3}(\sigma) &= (\sigma : y_i \mapsto genDelConfRes(\sigma(x_i), \sigma(indCap))),\\
f_{i,4}(\sigma) &= (\sigma : z_i \mapsto genDelRes(\sigma(x_i), \sigma(pCap)))
\end{aligned}
$$

The A-C formula for process I_{ld} is as follows:

$$\vdash \langle Ass^i_{ld},\ C^i_{ld}\rangle\ \{\varphi^i_{ld}\}\ I_{ld}\ \{\psi^i_{ld}\}$$

where the formal definition of φ^i_{ld}, ψ^i_{ld}, Ass^i_{ld}, and C^i_{ld} are as follows:

$$\varphi^i_{ld} \stackrel{\text{def}}{=} \#A = \#B = \#C = 0 \ \wedge \ |icTree| > 1$$

$$\psi^i_{ld} \stackrel{\text{def}}{=} false$$

$$\begin{aligned} Ass^i_{ld} \stackrel{\text{def}}{=} \ & \#A > 0 \rightarrow \\ & (isDelReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\ & isCurrentNode(last(A).snid) \wedge \\ & isCurrentNode(last(A).rnid)) \wedge \\ & isUniquePID(last(A).spid) \ \wedge \ isUniquePID(last(A).rpid))) \end{aligned}$$

$$\begin{aligned} C^i_{ld} \stackrel{\text{def}}{=} \ & (\#A = \#B > \#C \rightarrow \\ & \quad isDelConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap)) \wedge \\ & (\#A = \#B = \#C > 0 \rightarrow \\ & \quad isLdRes(last(C)) \ \wedge \ isValidCap(last(C).cap) \end{aligned}$$

The assertion network for I_{ld} is as follows:

$$Q_{s^i_{ld}} \stackrel{\text{def}}{=} \#A = \#B = \#C = 0 \ \wedge \ |icTree| > 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C \ \wedge \ |icTree| > 1$$

$$\begin{aligned} Q_{l_{i,2}} \stackrel{\text{def}}{=} \ & \#B = \#C = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| > 1 \wedge \\ & isDelReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\ & isCurrentNode(last(A).snid) \ \wedge \ isCurrentNode(last(A).rnid)) \wedge \\ & isUniquePID(last(A).spid) \ \wedge \ isUniquePID(last(A).rpid))) \end{aligned}$$

$$\begin{aligned} Q_{l_{i,3}} \stackrel{\text{def}}{=} \ & \#B = \#C = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| > 2 \wedge \\ & isValidInd(iCap)) \ \wedge \ isValidCap(pCap)) \end{aligned}$$

$$\begin{aligned} Q_{l_{i,4}} \stackrel{\text{def}}{=} \ & \#B = \#C = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| > 2 \wedge \\ & isValidCap(pCap)) \ \wedge \ isDelConfRes(y_i) \ \wedge \ isValidInd(y_i.cap) \end{aligned}$$

$$\begin{aligned} Q_{l_{i,5}} \stackrel{\text{def}}{=} \ & \#A = \#B = (\#C + 1) \ \wedge \ last(A) = x_i \ \wedge \ \ last(B) = y_i \wedge \\ & |icTree| > 2 \ \wedge \ isLdRes(z_i) \ \wedge \ isValidCap(z_i.cap)) \end{aligned}$$

$$\begin{aligned} Q_{l_{i,6}} \stackrel{\text{def}}{=} \ & \#A = \#B = (\#C + 1) \ \wedge \ last(A) = x_i \ \wedge \ \ last(B) = y_i \wedge \\ & |icTree| > 2 \ \wedge \ isLdRes(z_i) \ \wedge \ isValidCap(z_i.cap)) \end{aligned}$$

$$Q_{t^i_{ld}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{ld}, commitment C^i_{ld}, and channels A, B, and C.

- $\models Q_{s^i_{ld}} \rightarrow C^i_{ld}$ follows from the above definitions.

- $\models Q_{s^i_{ld}} \wedge Ass^i_{ld} \rightarrow Q_{l_{i,1}} \circ init^i_{ld}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \wedge Ass^i_{ld} \rightarrow ((Ass^i_{ld} \rightarrow Q_{l_{i,2}}) \wedge C^i_{ld}) \circ (f_{i,2} \circ f_{i,1} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i,\ h \mapsto v,\ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, communication through channel A just took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{ld} satisfies that the received message is a local-delegation request, and both the delegator and the receiver processes have unique PIDs. Thus, this verification holds.
- $\models Q_{l_{i,2}} \wedge Ass^i_{ld} \rightarrow Q_{l_{i,3}} \circ (f_{i,2} \circ f_{i,1})$. In this internal transition, function $f_{i,1}$ delegates the $x_i.cap$, creates a valid indicator capability pointing to the new i-cap inside the intermediate-capability tree, and assigns it to $indCap$. Hence, we have $\sigma' \models |icTree| > 2$. Furthermore, function $f_{i,2}$ creates a valid p-cap corresponding to $indCap$ and assigns it to $pCap$. Hence, this verification holds.
- $\models Q_{l_{i,3}} \wedge Ass^i_{ld} \rightarrow Q_{l_{i,4}} \circ f_{i,3}$. In this internal transition, function $f_{i,3}$ creates a delegation-confirmation response for the delegator process and assigns it to y_i, wherein $y_i.cap = indCap$. Hence, y_i contains a valid indicator capability. Thus, this verification holds.
- $\models Q_{l_{i,4}} \wedge Ass^i_{ld} \rightarrow ((Ass^i_{ld} \rightarrow Q_{l_{i,5}}) \wedge C^i_{ld}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_i)))$. In this output transition, program I_{ld} sends y_i through channel B. C^i_{ld} holds because $last(B) = \sigma(y_i)$ and $\sigma \models isLdConfRes(y_i) \wedge isValidInd(y_i.cap)$. Hence, this verification holds.
- $\models Q_{l_{i,5}} \wedge Ass^i_{ld} \rightarrow Q_{l_{i,6}} \circ f_{i,4}$. In this internal transition, function $f_{i,4}$ creates a delegation response for the receiver process and assigns it to z_i, wherein $z_i.cap = pCap$. Hence, z_i contains a valid process capability. Thus, this verification holds.
- $\models Q_{l_{i,6}} \wedge Ass^i_{ld} \rightarrow ((Ass^i_{ld} \rightarrow Q_{l_{i,1}}) \wedge C^i_{ld}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(z_i)))$. In this output transition, program I_{ld} sends z_i through channel C. C^i_{ld} holds because $\sigma \models isLdRes(z_i) \wedge isValidCap(z_i.cap)$. Hence, this verification holds.

Since $Q_{t^i_{ld}} \rightarrow \psi^i_{ld}$, the post-condition of program I_{ld} is satisfied.

□

5.6 Third Lemma: External Delegation

The Third lemma indicates that delegating a valid p-cap by the delegator process creates a valid p-cap for a process inside another node. This operation requires that both intermediate controllers and the resource controller participate. We formally define the lemma and proof it.

Lemma 3. *Given two processes with unique PIDs inside different process nodes, delegating a valid capability from the first process to the second process causes that the receiver and the delegator processes receive a valid p-cap and a valid indicator, respectively.*

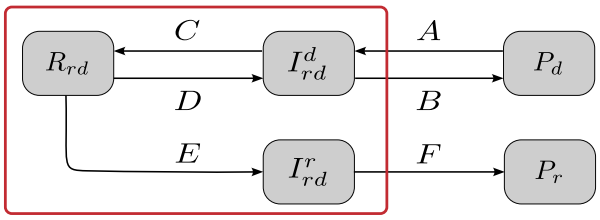


Figure 10: Syntactic Interfaces of the External-delegation Programs.

Proof. We use the assumption-commitment technique to prove the lemma.

To proof the lemma, we model the intermediate controller, operating at the delegator-process side, the resource controller, and the intermediate controller, operating at the receiver-process side, in a remote-delegation operation by programs $I^d_{rd} \stackrel{\text{def}}{=} (L^{i_d}_{rd}, T^{i_d}_{rd}, s^{i_d}_{rd}, t^{i_d}_{rd})$, $R_{rd} \stackrel{\text{def}}{=} (L^r_{rd}, T^r_{rd}, s^r_{rd}, t^r_{rd})$, and $I^r_{rd} \stackrel{\text{def}}{=} (L^{i_r}_{rd}, T^{i_r}_{rd}, s^{i_r}_{rd}, t^{i_r}_{rd})$, respectively. Figure 10 depicts these three programs and their syntactic interfaces. The messages are exchanged between the programs via the channels to handle remote-delegation requests. In addition, figures 11a, 11b, and 11c illustrate sequential diagrams of programs I^d_{rd}, R_{rd}, and I^r_{rd}, respectively. In the next step, we prove that the receiver process will receive a valid p-cap. In addition, we prove that process the delegator process receives a valid indicator capability.

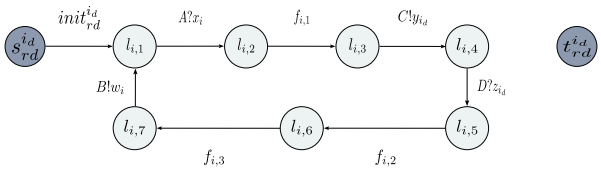
5.6.1 External Delegation - Delegating-Side Intermediate Controller

Program I^d_{rd} receives a remote-delegation request from the process via channel A, creates a corresponding request for program R_{rd}, and sends it through channel C. Furthermore, It receives the response from program R_{rd} via channel D. Program I^d_{rd} updates its capability tree when it receives an i-cap from the resource controller by inserting the indicator capability inside its tree. In addition, it creates the corresponding indicator capability for the inserted capability inside its tree. Finally, program I^d_{rd} creates a response for the process and sends the indicator inside the response to the process through channel B. The conditions and functions of I^d_{rd} are as follow:

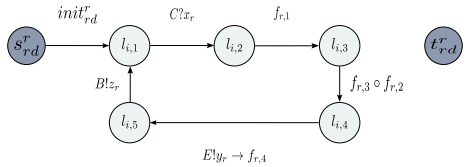
$$
\begin{aligned}
init^{i_d}_{rd}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : y_i \mapsto genRdReq(\sigma(rootPtr), \sigma(x_i)),\\
f_{i,2}(\sigma) &= (\sigma : pIndCap \mapsto insert(\sigma(icTree), \sigma(z_i))),\\
f_{i_d,3}(\sigma) &= (\sigma : w_i \mapsto genRdConfRes(\sigma(x), \sigma(pIndCap)))
\end{aligned}
$$

The A-C formula for process I^d_{rd} is as follows:

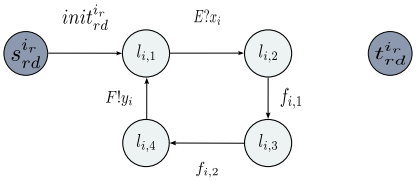
$$\vdash \langle Ass^{i_d}_{rd},\ C^{i_d}_{rd} \rangle\ \{\varphi^{i_d}_{rd}\}\ I^d_{rd}\ \{\psi^{i_d}_{rd}\}$$



(a) Program I^d_{rd}.



(b) Program R_{rd}.



(c) Program I^r_{rd}.

Figure 11: Remote-Delegation Processes.

where the formal definition of $\varphi^{i_d}_{rd}$, $\psi^{i_d}_{rd}$, $Ass^{i_d}_{rd}$, and $C^{i_d}_{rd}$ are as follows:

$$
\begin{array}{lcl}
\varphi^{i_d}_{rd} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| > 1 \\
\psi^{i_d}_{rd} & \stackrel{\text{def}}{=} & false \\
Ass^{i_d}_{rd} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & (isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\
 & & isUniquePID(last(A).spid) \wedge \\
 & & isUniquePID(last(A.rpid))) \wedge \\
 & & (\#C = \#D > 0 \rightarrow \\
 & & \quad isRdConfRes(last(D)) \ \wedge \ isValidInd(last(D).cap))) \wedge \\
C^{i_d}_{rd} & \stackrel{\text{def}}{=} & (\#A = \#B > 0 \rightarrow \\
 & & \quad isRdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap)) \wedge \\
 & & (\#C > 0 \rightarrow (isRdReq(last(C)) \ \wedge \ isValidCap(last(C).cap) \wedge \\
 & & \ isUniquePID(last(C).spid) \ \wedge \ isUniquePID(last(C).rpid)))
\end{array}
$$

The assertion network for I^d_{rd} is as follows:

$$Q_{s^{i_d}_{rd}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| > 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap)) \wedge isUniquePID(last(A).spid) \ \wedge \ isUniquePID(last(A).rpid)$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge isRdReq(y_i) \ \wedge \ isValidInd(y_i.cap)) \ \wedge \ isUniquePID(y_i.spid) \wedge isUniquePID(y_i.rpid)$$

$$Q_{l_{i,4}} \stackrel{\text{def}}{=} \#A = \#B = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge last(D) = z_i \ \wedge \ isRdConfRes(last(D)) \ \wedge \ isValidInd(last(D).cap)$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge last(D) = z_i \ \wedge \ isValidInd(pIndCap)$$

$$Q_{l_{i,7}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge last(D) = z_i \ \wedge \ isRdConfRes(w_i) \ \wedge \ isValidInd(w_i.cap)$$

$$Q_{i^{i_d}_{rd}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption $Ass^{i_d}_{rd}$, commitment $C^{i_d}_{rd}$, and channels A, B, C, and D.

- $\models Q_{s^{i_d}_{rd}} \rightarrow C^{i_d}_{rd}$ follows from the above definitions.
- $\models Q_{s^{i_d}_{rd}} \wedge Ass^{i_d}_{rd} \rightarrow Q_{l_{i,1}} \circ init^{i_d}_{rd}$. In this internal transition, just function $init^{i^d}_{rd}$ gets a pointer to the root of the capability tree. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ Ass^{i_d}_{rd} \rightarrow ((Ass^{i_d}_{rd} \rightarrow Q_{l_{i,2}}) \ \wedge \ C^{i_d}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of $Ass^{i_d}_{rd}$ satisfies it is a remote-delegation request, $last(A).cap$ is a valid indicator capability, and the delegator and receiver PIDs are unique. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ Ass^{i_d}_{rd} \rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ creates a remote-revocation request and assigns it to variable y_i, such that $isRdReq(y_i) = true$, $y_i.spid = last(A).spid$, and $y_i.rpid = last(A).rpid$. Hence, the delegator and receiver PIDs in the new request are unique because Ass^{i}_{ra} indicates that

both *PID*s inside the request from the process are unique. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge Ass^{i_d}_{rd} \rightarrow ((Ass^{i_d}_{rd} \rightarrow Q_{l_{i,7}}) \wedge C^{i_d}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I^d_{rd} sends y_i through channel C. $C^{i_d}_{rd}$ holds because $last(C) = \sigma(y_i)$, $isRdReq(y_i)$, and both $y_i.spid$ and $y_i.rpid$ are unique. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge Ass^{i_d}_{rd} \rightarrow ((Ass^{i_d}_{rd} \rightarrow Q_{l_{i,5}}) \wedge C^{i_d}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i,\ h \mapsto v,\ \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Since $\#D > 0$, $Ass^{i_d}_{rd}$ implies that $isRdConfRes(last(D))$ and $isValidInd(last(D).cap)$. Thus, this verification holds.

- $\models Q_{l_{i,5}} \wedge Ass^{i_d}_{rd} \rightarrow Q_{l_{i,6}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ inserts the received intermediate indicator into the capability tree and returns $pIndCap$ that points to it inside the tree. We have $isValidCap(pIndCap)$ because $ValidInd(last(D).cap)$, and $pIndCap$ points to the it after the intermediate controller inserts it inside the tree. Thus, this verification holds.

- $\models Q_{l_{i,6}} \wedge Ass^{i_d}_{rd} \rightarrow Q_{l_{i,7}} \circ (f_{i,3} \circ f_{i,2})$. In this internal transition, function $f_{i,3}$ creates a remote-delegation confirmation response for the process and assigns it to w_i, wherein $w_i.cap = pIndCap$. Thus, this verification holds.

- $\models Q_{l_{i,7}} \wedge Ass^{i_d}_{rd} \rightarrow ((Ass^{i_d}_{rd} \rightarrow Q_{l_{i,1}}) \wedge C^{i_d}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I^d_{rd} sends w_i through channel B. $C^{i_d}_{rd}$ holds because $\sigma \models isRdConfRes(w_i) \wedge isValidInd(w_i.cap)$. Hence, this verification holds.

Since $Q_{t^{i_d}_{rd}} \rightarrow \psi^{i_d}_{rd}$, the post-condition of program I^d_{rd} is satisfied.

5.6.2 External Delegation - Resource Controller

Program R_{rd} receives a remote-delegation request from program I^d_{rd} via channel C. In the next step, program R_{rd} creates creates the delegated capability and inserts it inside the capability tree. Furthermore, it creates the delegated resource capability's corresponding intermediate and indicator capabilities. Afterward, program R_{rd} creates the remote-delegation and confirmation responses and inserts the intermediate and indicator capabilities inside them. Finally, it sends the responses through channels E and D toward programs I^r_{rd} and I^d_{rd}, respectively. The functions of R_{rd} are as follows:

$$
\begin{aligned}
init^r_{rd}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(rcTree))),\\
f_{r,1}(\sigma) &= (\sigma : indCap \mapsto delegate(\sigma(rootPtr), \sigma(x_r))),\\
f_{r,2}(\sigma) &= (\sigma : iCap \mapsto genCapFromInd(\sigma(indCap), \sigma(indCap))),\\
f_{r,3}(\sigma) &= (\sigma : y_r \mapsto genRdRes(\sigma(x_r), \sigma(iCap))),\\
f_{r,3}(\sigma) &= (\sigma : z_r \mapsto genRdConfRes(\sigma(x_r), \sigma(indCap))),
\end{aligned}
$$

where $rcTree$ is the resource-capability tree, $indCap$ is an intermediate indicator, and $iCap$ is an intermediate capability.

The A-C formula for process R_{rd} is as follows:

$$\vdash \langle Ass^r_{rd}\ C^r_{rd}\rangle\ \{\varphi^r_{rd}\}\ R_{rd}\ \{\psi^r_{rd}\}$$

where the formal definition of φ_r, ψ_r , Ass_r, and C_r are as follows:

$$\begin{array}{lcl}
\varphi^r_{rd} & \stackrel{\text{def}}{=} & \#C = \#D = \#E = 0\ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{rd} & \stackrel{\text{def}}{=} & false \\
Ass^r_{rd} & \stackrel{\text{def}}{=} & (\#C > 0 \rightarrow \\
 & & \quad (isRdReq(last(C))\ \wedge\ isValidInd(last(C).cap) \wedge \\
 & & \quad\ isUniquePID(last(C).spid) \wedge \\
 & & \quad\ isUniquePID(last(C).rpid))) \\
C^r_{rd} & \stackrel{\text{def}}{=} & (\#C = \#D > 0 \rightarrow \\
 & & \quad (isRdConfRes(last(D))\ \wedge\ isValidInd(last(D).cap))) \wedge \\
 & & (\#C = \#E > 0 \rightarrow \\
 & & \quad (isRdRes(last(E))\ \wedge\ isValidCap(last(E).cap)))
\end{array}$$

The assertion network for R_{rd} is as follows:

$$\begin{array}{lcl}
Q_{s^r_{rd}} & \stackrel{\text{def}}{=} & \#C = \#D = \#E = 0\ \wedge\ |rcTree| \geqslant 1 \\
Q_{r_1} & \stackrel{\text{def}}{=} & \#C = \#D = \#E \\
Q_{r_2} & \stackrel{\text{def}}{=} & \#D = \#E = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ isRdReq(last(C)) \wedge \\
 & & isValidInd(last(C).cap)\ \wedge\ isUniquePID(last(C).spid) \wedge \\
 & & isUniquePID(last(C).rpid) \\
Q_{r_3} & \stackrel{\text{def}}{=} & \#D = \#E = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ isValidInd(indCap) \\
Q_{r_4} & \stackrel{\text{def}}{=} & \#D = \#E = (\#C - 1)\ \wedge\ last(C) = x_r\ \wedge\ isValidInd(indCap) \wedge \\
 & & isRdRes(y_r)\ \wedge\ isValidCap(y_r.cap) \\
Q_{r_5} & \stackrel{\text{def}}{=} & \#C = \#E = (\#D + 1)\ \wedge\ last(C) = x_r\ \wedge\ last(E) = y_r \wedge \\
 & & isRdConfRes(z_r)\ \wedge\ isValidInd(z_r.cap) \\
Q_{t^r_{rd}} & \stackrel{\text{def}}{=} & false
\end{array}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^r_{rd}, commitment C^r_{rd}, and channels C, D, and E.

- $\models Q_{s^r_{rd}} \rightarrow C^r_{rd}$ follows from the above definitions.
- $\models Q_{s^r_{rd}} \wedge Ass^r_{rd} \rightarrow Q_{l_{r,1}} \circ init^r_{rd}$. In this internal transition, just function $init^r_{rd}$ gets a pointer to the root of the capability tree. Hence, this verification holds.

- $\models Q_{l_{r,1}} \wedge Ass^r_{rd} \rightarrow ((Ass^r_{rd} \rightarrow Q_{r_2}) \wedge C^r_{rd}) \circ$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r,\ h \mapsto v,\ \sigma(h).(C, v))$ for arbitrary v. In this input transition, a communication through channel C just took place. Function g assigns the last received value from channel C to x_r. Since $\#C > 0$, the first implication of Ass^r_{rd} satisfies $isRdReq(last(C))$, $isUniquePID(last(C).spid)$, $isUniquePID(last(C).rpid)$, and $isValidInd(last(C).cap)$ predicates. C^r_{rd} holds since neither $\#C = \#D$ nor $\#C = \#E$. Thus, this verification holds.
- $\models Q_{l_{r,2}} \wedge Ass^r_{rd} \rightarrow Q_{l_{r,3}} \circ (unlockTree \circ f_{r,1})$. In this internal transition, function $f_{r,1}$ delegates the capability and inserts the newly generated capability into the capability tree. Furthermore, function $f_{r,1}$ returns a valid indicator capability that points to the inserted resource capability inside the tree. Variable $indCap$ keeps the returned intermediate indicator. Hence, we have $\sigma' \models isValidInd(indCap)$. Thus, this verification holds.
- $\models Q_{l_{r,3}} \wedge Ass^r_{rd} \rightarrow Q_{l_{r,4}} \circ (f_{r,3} \circ f_{r,2})$. In this internal transition, function $f_{r,2}$ generates a intermediate capability from the newly generated resource capability and assign it to variable $iCap$. Hence, we have $isValidCap(iCap)$. Afterward, function $f_{r,3}$ creates remote-delegation response and inserts $iCap$ as its capability. Thus, this verification holds.
- $\models Q_{l_{r,4}} \wedge Ass^r_{rd} \rightarrow ((Ass^r_{rd} \rightarrow Q_{l_{r,5}}) \wedge C^r_{rd}) \circ (f_{r,3} \circ g)$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(E, \sigma(y_r)))$. In this output transition, program R_{rd} sends y_r through channel E. Commitment C^r_{rd} holds because we have $isRdRes(y_r)$ and $isValidCap(y_r.cap)$. Afterward, function $f_{r,3}$ creates a remote-delegation confirmation response, inserts $indCap$ as its capability, and assigns it to variable z_r. Hence, it satisfies both $isRdConfRes(z_r)$ and $isValidInd(z_r.cap)$. Thus, this verification holds.
- $\models Q_{l_{r,5}} \wedge Ass^r_{rd} \rightarrow ((Ass^r_{rd} \rightarrow Q_{l_{r,1}}) \wedge C^r_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(y_r)))$. In this output transition, program R_{rd} sends z_r through channel B. Commitment C^r_{rd} holds because we have $isRdConfRes(z_r)$ and $isValidInd(z_r.cap)$. Thus, this verification holds.

Since $Q_{t^r_{rd}} \rightarrow \psi^r_{rd}$, the post-condition of program R_{rd} is satisfied.

5.6.3 External Delegation - Receiving-Side Intermediate Controller

Program I^r_{rd} receives a remote-delegation request from the process via channel E. Then, it updates its capability tree by inserting the received intermediate capability inside the tree. In addition, it creates the corresponding p-cap for the inserted i-cap. Finally, program I^r_{rd} creates a remote-delegation response for the receiving process and sends the p-cap inside the response to the process through channel B. The functions of I^r_{rd} are as follows:

$$
\begin{aligned}
init^{i_r}_{rd}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : pCap \mapsto insert(\sigma(rootPtr), \sigma(x_i.cap)))\\
f_{i,2}(\sigma) &= (\sigma : y_i \mapsto genRdRes(\sigma(x_i), \sigma(pCap))),
\end{aligned}
$$

The A-C formula for process I^d_{rd} is as follows:

$$\vdash \langle Ass^{i_r}_{rd},\ C^{i_r}_{rd}\rangle\ \{\varphi^{i_r}_{rd}\}\ I^r_{rd}\ \{\psi^{i_r}_{rd}\}$$

where the formal definition of $\varphi^{i_r}_{rd}$, $\psi^{i_r}_{rd}$, $Ass^{i_r}_{rd}$, and $C^{i_r}_{rd}$ are as follows:

$$\begin{aligned}
\varphi^{i_r}_{rd} &\stackrel{\text{def}}{=} & \#E = \#F = 0 \ \wedge\ |icTree| \geqslant 1\\
\psi^{i_r}_{rd} &\stackrel{\text{def}}{=} & false\\
Ass^{i_r}_{rd} &\stackrel{\text{def}}{=} & \#E > 0 \rightarrow\\
& & (isRdRes(last(E)) \ \wedge\ isValidCap(last(E).cap)\\
C^{i_r}_{rd} &\stackrel{\text{def}}{=} & \#E = \#F > 0 \rightarrow\\
& & isRdRes(last(F)) \ \wedge\ isValidCap(last(F).cap)
\end{aligned}$$

The assertion network for I^r_{rd} is as follows:

$$\begin{aligned}
Q_{s^{i_r}_{rd}} &\stackrel{\text{def}}{=} \#E = \#F = 0 \ \wedge\ |icTree| \geqslant 1\\
Q_{l_{i,1}} &\stackrel{\text{def}}{=} \#E = \#F\\
Q_{l_{i,2}} &\stackrel{\text{def}}{=} \#F = (\#E - 1) \ \wedge\ last(E) = x_i \ \wedge\\
& \quad isRdRes(last(E)) \ \wedge\ isValidCap(last(E).cap))\\
Q_{l_{i,3}} &\stackrel{\text{def}}{=} \#F = (\#E - 1) \ \wedge\ last(E) = x_i \ \wedge\\
& \quad isValidCap(pCap)\\
Q_{l_{i,4}} &\stackrel{\text{def}}{=} \#F = (\#E - 1) \ \wedge\ last(E) = x_i \ \wedge\\
& \quad isRdRes(y_i) \ \wedge\ isValidCap(y_i.cap))\\
Q_{t^{i_d}_{rd}} &\stackrel{\text{def}}{=} false
\end{aligned}$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption $Ass^{i_r}_{rd}$, commitment $C^{i_r}_{rd}$, and channels E and F.

- $\models Q_{s^{i_r}_{rd}} \rightarrow C^{i_r}_{rd}$ follows from the above definitions.
- $\models Q_{s^{i^r}_{rd}} \wedge Ass^{i_r}_{rd} \rightarrow Q_{l_{i,1}} \circ init^{i^r}_{rd}$. In this internal transition, just function $init^{i^r}_{rd}$ gets a pointer to the root of the capability tree. Hence, this verification holds.
- $\models Q_{l_{i,1}} \wedge Ass^{i_r}_{rd} \rightarrow ((Ass^{i_r}_{rd} \rightarrow Q_{l_{i,2}}) \wedge C^{i_r}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i,\ h \mapsto v,\ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of $Ass^{i_r}_{rd}$ satisfies that the last received message is a remote-delegation response,and the capability inside it is a valid intermediate capability. Thus, this verification holds.

- $\models Q_{l_{i,2}} \wedge Ass^{i_r}_{rd} rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ inserts the received intermediate capability into the capability tree and returns $pCap$ that points to it inside the tree. Hence, we have $isValidCap(pCap)$. Thus, this verification holds.
- $\models Q_{l_{i,3}} \wedge Ass^{i_r}_{rd} \rightarrow Q_{l_{i,4}} \circ (f_{i,2} \circ unlockTree)$. In this internal transition, function $f_{i,2}$ creates a remote-delegation response for the process and assigns it to y_i, wherein $y_i.cap = pCap$. Hence, this verification holds.
- $\models Q_{l_{i,4}} \wedge Ass^{i_r}_{rd} \rightarrow ((Ass^{i_r}_{rd} \rightarrow Q_{l_{i,1}}) \wedge C^{i_r}_{rd}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(F, \sigma(y_i)))$. In this output transition, program I^r_{rd} sends y_i through channel F. $C^{i_r}_{rd}$ holds because $\sigma \models isRdRes(y_i) \wedge isValidCap(y_i.cap)$. Thus, this verification holds.

Since $Q_{t^{i_r}_{rd}} \rightarrow \psi^{i_r}_{rd}$, the post-condition of program I^r_{rd} is satisfied.

5.6.4 External Delegation - Parallel Composition

This section presents the parallel composition of the remote-delegation components based on the described rule in 1. Consider the parallel composition $R_{rd} \parallel I^d_{rd} \parallel I^r_{rd}$. Program R_{rd} satisfies the assumption-commitment pair $(Ass^r_{rd},\ C^r_{rd})$ as follows:

$$\vdash \langle Ass^r_{rd}\ C^r_{rd} \rangle\ \{\varphi^r_{rd}\}\ R_{rd}\ \{\psi^r_{rd}\}$$

where the formal definition of φ_r, ψ_r , Ass_r, and C_r are as follows:

$$\begin{aligned}
\varphi^r_{rd} &\stackrel{\text{def}}{=} \#C = \#D = \#E = 0 \ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{rd} &\stackrel{\text{def}}{=} false \\
Ass^r_{rd} &\stackrel{\text{def}}{=} (\#C > 0 \rightarrow \\
&\quad (isRdReq(last(C)) \ \wedge\ isValidInd(last(C).cap) \wedge \\
&\quad\ isUniquePID(last(C).spid) \wedge \\
&\quad\ isUniquePID(last(C).rpid))) \\
C^r_{rd} &\stackrel{\text{def}}{=} (\#C = \#D > 0 \rightarrow \\
&\quad (isRdConfRes(last(D)) \ \wedge\ isValidInd(last(D).cap))) \wedge \\
&\quad (\#C = \#E > 0 \rightarrow \\
&\quad (isRdRes(last(E)) \ \wedge\ isValidCap(last(E).cap)))
\end{aligned}$$

Program I^d_{rd} satisfies the assumption-commitment pair $(Ass^{i_d}_{rd},\ C^{i_d}_{rd})$ as follows:

$$\vdash \langle Ass^{i_d}_{rd},\ C^{i_d}_{rd} \rangle\ \{\varphi^{i_d}_{rd}\}\ I^d_{rd}\ \{\psi^{i_d}_{rd}\}$$

where the formal definition of $\varphi_{rd}^{i_d}$, $\psi_{rd}^{i_d}$, $Ass_{rd}^{i_d}$, and $C_{rd}^{i_d}$ are as follows:

$$
\begin{array}{lcl}
\varphi_{rd}^{i_d} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree_d| > 1 \\
\psi_{rd}^{i_d} & \stackrel{\text{def}}{=} & false \\
Ass_{rd}^{i_d} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & (isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \wedge \\
 & & isUniquePID(last(A).spid) \wedge \\
 & & isUniquePID(last(A.rpid))) \wedge \\
 & & (\#C = \#D > 0 \rightarrow \\
 & & \quad isRdConfRes(last(D)) \ \wedge \ isValidInd(last(D).cap))) \wedge \\
C_{rd}^{i_d} & \stackrel{\text{def}}{=} & (\#A = \#B > 0 \rightarrow \\
 & & \quad isRdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap)) \wedge \\
 & & (\#C > 0 \rightarrow \\
 & & (isRdReq(last(C)) \ \wedge \ isValidCap(last(C).cap) \wedge \\
 & & isUniquePID(last(C).spid) \ \wedge \ isUniquePID(last(C).rpid)))
\end{array}
$$

Program I_{rd}^{r} satisfies the assumption-commitment pair $(Ass_{rd}^{i_r}, \ C_{rd}^{i_r})$ as follows:

$$\vdash \langle Ass_{rd}^{i_r}, \ C_{rd}^{i_r} \rangle \ \{\varphi_{rd}^{i_r}\} \ I_{rd}^{r} \ \{\psi_{rd}^{i_r}\}$$

where the formal definition of $\varphi_{rd}^{i_r}$, $\psi_{rd}^{i_r}$, $Ass_{rd}^{i_r}$, and $C_{rd}^{i_r}$ are as follows:

$$
\begin{array}{lcl}
\varphi_{rd}^{i_r} & \stackrel{\text{def}}{=} & \#E = \#F = 0 \ \wedge \ |icTree_r| \geqslant 1 \\
\psi_{rd}^{i_r} & \stackrel{\text{def}}{=} & false \\
Ass_{rd}^{i_r} & \stackrel{\text{def}}{=} & \#E > 0 \rightarrow \\
 & & (isRdRes(last(E)) \ \wedge \ isValidCap(last(E).cap) \\
C_{rd}^{i_r} & \stackrel{\text{def}}{=} & \#E = \#F > 0 \rightarrow \\
 & & \quad isRdRes(last(F)) \ \wedge \ isValidCap(last(F).cap)
\end{array}
$$

By applying the parallel composition rule, we deduce as follows:

$$
\begin{array}{c}
\vdash \langle Ass_{rd}, \ C_{rd} \rangle \\
\{\varphi_{rd}\} \ R_{ra} \parallel I_{rd}^{d} \parallel I_{rd}^{r} \ \{\psi_{rd}\}
\end{array}
$$

where the formal definition of φ_{rd}, ψ_{rd}, Ass_{rd}, and C_{rd} are as follows:

$$
\begin{aligned}
\varphi_{rd} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = \#E = \#F = 0 \ \wedge \ |rcTree| \geqslant 1 \ \wedge \\
&\quad |icTree_d| \geqslant 1 \ \wedge \ |icTree_r| \geqslant 1 \\
\psi_{rd} &\stackrel{\text{def}}{=} false \\
Ass_{rd} &\stackrel{\text{def}}{=} \#A > 0 \rightarrow \\
&\quad (isRdReq(last(A)) \ \wedge \ isValidCap(last(A).cap) \ \wedge \\
&\quad isUniquePID(last(A).spid) \ \wedge \\
&\quad isUniquePID(last(A).rpid)) \\
C_{rd} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = \#E = \#F > 0 \rightarrow \\
&\quad (isRdConfRes(last(B)) \ \wedge \ isValidInd(last(B).cap) \\
&\quad (isRdRes(last(F)) \ \wedge \ isValidCap(last(F).cap))
\end{aligned}
$$

□

5.7 Fourth Lemma: External Revocation

The fourth lemma is about the remote-revocation operation. We first describe the remote revocation instead of the local revocation because a process may re-delegate a locally delegated capability to a remote process. Hence, a local-revocation operation may implicitly include several remote-revocation operations. The fourth lemma indicates that the remote revocation of a valid p-cap by the delegator process will invalidate the delegated p-cap owned by another process inside another node. Since it is a remote-revocation operation, the intermediate controller in the delegator-process side and the resource controller handle the revocation together We formally define the lemma and proof it.

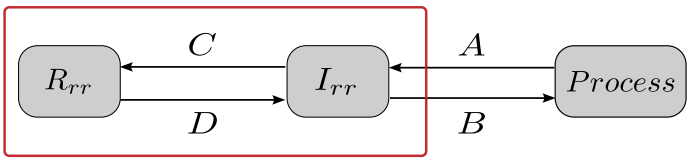
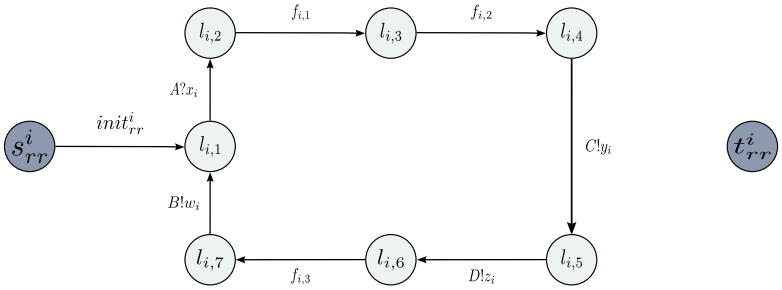


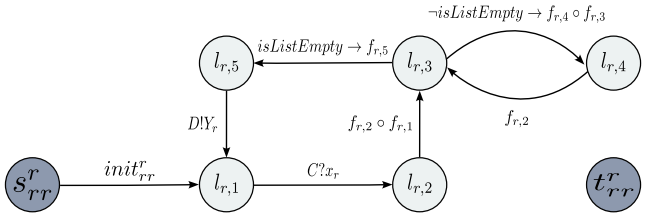
Figure 12: Syntactic Interfaces of the Remote-revocation Programs.

Lemma 4. *Given a process with a unique PID in a process node, an intermediate controller in the same node, and a resource controller in a resource node, revoking a delegated p-cap, which points to a resource inside the resource node, to another process inside another node invalidates the delegated p-cap.*

Proof. We use the assumption-commitment technique to prove the lemma.



(a) Program I_{rr}.



(b) Program R_{rr}.

Figure 13: Remote Revocation Programs.

We model the intermediate controller, operating at the delegator-process side, and the resource controller in a remote-revocation operation by programs $I_{rr} \stackrel{\text{def}}{=} (L^i_{rr}, T^i_{rr}, s^i_{rr}, t^i_{rr})$ and $R_{rr} \stackrel{\text{def}}{=} (L^r_{rr}, T^r_{rr}, s^r_{rr}, t^r_{rr})$, respectively. The programs and the process exchange messages via synchronous channels to handle remote-revocation requests. Figure 12 depicts these two programs and their syntactic interfaces. Furthermore, figures 13a and 13b illustrate sequential diagrams of programs I_{rr} and R_{rr}, respectively. In the next step, we prove that when the process sends a remote-revocation request, programs I_{rr} and R_{rr} revoke the delegated capability and all its re-delegated capabilities.

We need to define a loop-invariant for program I_{rr} and a loop-invariant for program R_{rr} to prove the correctness of the lemma. When an intermediate controller conducts a remote-delegation operation, it inserts an indicator capability as the child of the delegated capability inside its capability tree. There would be no sub-tree after the inserted capability inside the tree because of its indicator type. On the other hand, the resource controller inserts a r-cap inside its capability tree. Hence, the capability may have a sub-tree because of the re-delegation(s).

5.7.1 External Revocation - Intermediate Controller

Program I_{rr} receives a remote-revocation request from the process via channel A. The request contains a valid process indicator that points to an intermediate indicator inside the intermediate-capability tree. In the next step, program I_{rr} copies the intermediate indicator and removes it from the tree afterward. Furthermore, it creates a correspond-

ing remote-revocation request and sends it through channel C to program R_{rr}. Finally, when program I_{rr} receives the response from program R_{rr} via channel D, it creates a response for the process and sends the response toward the process through channel B. The functions of I_{rr} are as follow:

$$\begin{aligned}
init^i_{rr}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{i,1}(\sigma) &= (\sigma : iIndCap \mapsto revoke(\sigma(rootPtr), \sigma(x_i.cap))),\\
f_{i,2}(\sigma) &= (\sigma : y_i \mapsto genRrReq(\sigma(x_i), \sigma(iIndCap))),\\
f_{i,3}(\sigma) &= (\sigma : w_i \mapsto genRevConfRes(\sigma(z_i)))
\end{aligned}$$

The A-C formula for program I_{rr} is as follows:

$$\vdash \langle Ass^i_{rr},\ C^i_{rr}\rangle\ \{\varphi^i_{rr}\}\ I_{rr}\ \{\psi^i_{rr}\}$$

where the formal definition of φ^i_{rr}, ψ^i_{rr}, Ass^i_{rr}, and C^i_{rr} are as follows:

$$\begin{aligned}
\varphi^i_{rr} &\stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1\\
\psi^i_{rr} &\stackrel{\text{def}}{=} false\\
Ass^i_{rr} &\stackrel{\text{def}}{=} (\#A > 0 \rightarrow\\
&\quad isRrReq(last(A))\ \wedge\ isUniquePID(last(A).spid)))\\
&\quad isValidInd(last(A).cap)) \wedge\\
&\quad (\#C = \#D > 0 \rightarrow isRrConfRes(last(D)))\\
C^i_{rr} &\stackrel{\text{def}}{=} (\#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))) \wedge\\
&\quad (\#C > 0 \rightarrow isRrConfRes(last(C)))
\end{aligned}$$

The assertion network for I_{rr} is as follows:

$$Q_{s^i_{rr}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree \geqslant 2$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D \ \wedge \ |icTree| \geqslant 2$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| > 2 \wedge isRrReq(last(A)) \ \wedge \ isUniquePID(last(A).spid) \wedge isValidInd(last(A).cap)$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| \geqslant 2 \wedge isValidInd(iIndCap) \ \wedge \ isTreeLocked$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ |icTree| \geqslant 2 \wedge isRrReq(y_i) \ \wedge \ isUniquePID(y_i.spid) \ \wedge \ isValidInd(y_i.cap)$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge |icTree| \geqslant 2$$

$$Q_{l_{i,7}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge last(D) = z_i \ \wedge \ isRrConfRes(z_i) \ \wedge \ |icTree| \geqslant 2$$

$$Q_{l_{i,8}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \wedge last(D) = z_i \ \wedge \ |icTree| \geqslant 2 \ \wedge \ isRrConfRes(w_i)$$

$$Q_{t^i_{rr}} \stackrel{\text{def}}{=} false$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{rr}, commitment C^i_{rr}, and channels A, B, C, and D.

- $\models Q_{s^i_{rr}} \rightarrow C^i_{rr}$ follows from the above definitions.
- $\models Q_{s^i_{rr}} \ \wedge \ Ass^i_{rr} \ \rightarrow \ Q_{l_{i,1}} \circ init^i_{rr}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ Ass^i_{rr} \rightarrow ((Ass^i_{rr} \rightarrow Q_{l_{i,2}}) \ \wedge \ C^i_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{rr} satisfies $isRrReq(last(A))$, $isUniquePID(last(A).spid)$, and $isValidInd(last(A).cap)$. Furthermore, we have $\sigma' \models |icTree| > 2$ because $isValidInd(last(A).cap)$ indicates that there is an intermediate indicator in addition to the root and the parent capability inside the tree. Thus, this verification holds.
- $\models Q_{l_{i,2}} \ \wedge \ Ass^i_{rr} \rightarrow Q_{l_{i,3}} \circ f_{i,1}$. In this internal transition, function $f_{i,1}$ copies the intermediate indicator inside the tree, which $last(A).cap$ points to it, removes it from the tree, and assigns the copied intermediate indicator to variable $iIndCap$ when it returns. Thus, this verification holds.

- $\models Q_{l_{i,3}} \wedge Ass^i_{rr} \rightarrow Q_{l_{i,4}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ creates a capability-revocation request for program R_{rr} and assigns it to variable y_i, such that $isRrReq(y_i) = true$ and $y_i.spid = last(A).spid$. Furthermore, we have $isUniquePID(y_i.spid) = true$ because $isUniquePID(last(A).spid) = true$ and $y_i.spid = last(A).rpid$. In addition, this transition satisfies $isValidInd(y_i.cap)$ because we have $isValidInd(indCap)$ and $y_i.cap = indCap$. Thus, this verification holds.
- $\models Q_{l_{i,4}} \wedge Ass^i_{rr} \rightarrow ((Ass^i_{rr} \rightarrow Q_{l_{i,5}}) \wedge C^i_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{rr} sends y_i through channel C. C^i_{rr} holds because $last(C) = \sigma(y_i)$ and $isRrReq(y_i)$, $isUniquePID(y_i.spid)$, and $isValidInd(y_i.cap)$. Hence, this verification holds.
- $\models Q_{l_{i,5}} \wedge Ass^i_{rr} \rightarrow ((Ass^i_{rr} \rightarrow Q_{l_{i,6}}) \wedge C^i_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i,\ h \mapsto v,\ \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Since $\#D > 0$, Ass^i_{rr} implies that $\sigma' \models isRrConfRes(z_i)$. Thus, this verification holds.
- $\models Q_{l_{i,6}} \wedge Ass^i_{rr} \rightarrow Q_{l_{i,7}} \circ f_{i,3}$. In this internal transition, function $f_{i,3}$ creates a remote revocation-confirmation response for the process and assigns it to w_i such that $isRrConfRes(w_i)$. Hence, this verification holds.
- $\models Q_{l_{i,7}} \wedge Ass^i_{rr} \rightarrow ((Ass^i_{rr} \rightarrow Q_{l_{i,1}}) \wedge C^i_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{rr} sends w_i through channel B. Commitment C^i_{rr} holds because $\sigma \models isRrConfRes(w_i)$. Hence, this verification holds.

Since $Q_{l^i_{rr}} \rightarrow \psi^i_{rr}$, the post-condition of program I_{rr} is satisfied.

5.7.2 External Revocation - Resource Controller

Program R_{rr} receives a capability-revocation request from program I_{rr} via channel B. In the next step, program R_{rr} removes the delegated capability and all re-delegated capabilities in its sub-tree. Finally, it sends a revocation-confirmation response toward program I_{rr}. The functions of R_{rr} are defined as follow:

$$
\begin{aligned}
init^r_{rr}(\sigma) &= (\sigma : rootPtr \mapsto getRoot(\sigma(icTree))),\\
f_{r,1}(\sigma) &= (\sigma : subtreeList \mapsto revoke(\sigma(rootPtr), \sigma(x_r.cap))),\\
f_{r,2}(\sigma) &= (\sigma : isListEmpty \mapsto checkList(\sigma(subtreeList))),\\
f_{r,3}(\sigma) &= (\sigma : nextCap \mapsto getNextCap(\sigma(subtreeList))),\\
f_{r,4}(\sigma) &= (\sigma : revokeCap(\sigma(nextCap))),\\
f_{r,5}(\sigma) &= (\sigma : y_r \mapsto genRevConfRes(\sigma(x_r)))
\end{aligned}
$$

The A-C formula for program R_{rr} is as follows:

$$\vdash \langle Ass^r_{rr},\ C^r_{rr}\rangle\ \{\varphi^r_{rr}\}\ R_{rr}\ \{\psi^r_{rr}\}$$

where the formal definition of φ^r_{rr}, ψ^r_{rr}, Ass^r_{rr}, and C^r_{rr} are as follows:

$$
\begin{aligned}
\varphi^r_{rr} &\stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge \ |rcTree| \geqslant 1 \\
\psi^r_{rr} &\stackrel{\text{def}}{=} false \\
Ass^r_{rr} &\stackrel{\text{def}}{=} \#C > 0 \rightarrow \\
&\quad (isRrReq(last(B)) \ \wedge \ isUniquePID(last(B).spid) \wedge \\
&\quad isValidInd(last(B).cap)) \\
C^r_{rr} &\stackrel{\text{def}}{=} \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{aligned}
$$

The assertion network for R_{rr} is as follows:

$$
\begin{aligned}
Q_{s^r_{rr}} &\stackrel{\text{def}}{=} \#C = \#D = 0 \ \wedge \ |rcTree \geqslant 2 \\
Q_{l_{r,1}} &\stackrel{\text{def}}{=} \#C = \#D \ \wedge \ |rcTree| \geqslant 2 \\
Q_{l_{r,2}} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ last(C) = x_r \ \wedge \ |rcTree| > 2 \wedge \\
&\quad isRrReq(last(C)) \ \wedge \ isUniquePID(last(C).spid) \wedge \\
&\quad isValidInd(last(C).cap) \\
Q_{l_{r,3}} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ last(C) = x_r \ \wedge \ |rcTree| \geqslant 2 \\
Q_{l_{r,4}} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ last(C) = x_r \ \wedge \ |rcTree| \geqslant 2 \ \wedge \ \neg isListEmpty \\
Q_{l_{r,5}} &\stackrel{\text{def}}{=} \#D = (\#C - 1) \ \wedge \ last(C) = x_r \ \wedge \ |rcTree| \geqslant 2 \ \wedge \ isListEmpty \wedge \\
&\quad isRrConfRes(y_r) \\
Q_{tr^r_{rr}} &\stackrel{\text{def}}{=} false
\end{aligned}
$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^r_{rr}, commitment C^r_{rr}, and channels C and D.

- $\models Q_{s^r_{rr}} \rightarrow C^r_{rr}$ follows from the above definitions.
- $\models Q_{s^r_{rr}} \ \wedge \ Ass^r_{rr} \ \rightarrow \ Q_{l_{r,1}} \circ init^r_{rr}$. In this internal transition, the size of the intermediate-capability tree does not change. Hence, this verification holds.
- $\models Q_{l_{r,1}} \ \wedge \ Ass^r_{rr} \rightarrow ((Ass^r_{rr} \rightarrow Q_{l_{r,2}}) \ \wedge \ C^r_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_r, \ h \mapsto v, \ \sigma(h).(C, v))$ for arbitrary value v. In this input transition, just communication through channel C took place, and function g assigns the received value to x_r. Since $\#C > 0$, the first implication of Ass^r_{rr} satisfies $isRrReq(last(C))$, $isUniquePID(last(C).spid)$, and $isValidInd(last(C).cap)$. Furthermore, we have $\sigma' \models |rcTree| > 2$ because $isValidInd(last(C).cap)$ indicates that there is a delegated resource capability in addition to the root and the parent capability inside the tree. Thus, this verification holds.
- $\models Q_{l_{r,2}} \ \wedge \ \neg isListEmpty \ \wedge \ Ass^r_{rr} \rightarrow Q_{l_{r,3}} \circ (f_{r,2} \circ f_{r,1})$. In this internal transition, function $f_{i,1}$ revokes the resource capability inside the tree, which $last(C).cap$

points to it. In addition, it returns the sub-tree of the resource capability as a list. Thus, we have $\sigma' \models |rcTree| \geqslant 2$. In the next step, function $f_{r,2}$ checks if the list is empty. Hence, this verification holds.

- $\models Q_{l_{r,3}} \wedge \neg isListEmpty \wedge Ass^r_{rr} \rightarrow Q_{l_{r,4}} \circ (f_{r,4} \circ f_{r,3})$. In this internal transition, the $\neg isListEmpty$ condition implies that $\sigma' \models \neg isListEmpty$. Hence, function $f_{i,3}$ gets the next resource capability that program R_{rr} should revokes . It assigns the next resource capability to variable $nextCap$. Thus, this verification holds.
- $\models Q_{l_{r,4}} \wedge Ass^r_{rr} \rightarrow Q_{l_{r,3}} \circ f_{r,2}$. In this internal transition, function $f_{r,2}$ checks if the list is empty. Hence, this verification holds.
- $\models Q_{l_{r,3}} \wedge isListEmpty \wedge Ass^r_{rr} \rightarrow Q_{l_{r,5}} \circ f_{r,5}$. In this internal transition, the $isListEmpty$ condition implies that $\sigma' \models isListEmpty$. Hence, function $f_{r,5}$ generates a revocation-confirmation response and assigns it to variable y_r. Hence, this verification holds.
- $\models Q_{l_{r,5}} \wedge Ass^r_{rr} \rightarrow ((Ass^r_{rr} \rightarrow Q_{l_{r,1}}) \wedge C^r_{rr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(D, \sigma(y_r)))$. In this output transition, program R_{rr} sends y_r through channel D. Commitment C^r_{rr} holds because $\sigma \models isRrConfRes(y_r)$. Hence, this verification holds.

Since $Q_{t^r_{rr}} \rightarrow \psi^r_{rr}$, the post-condition of program R_{rr} is satisfied.

5.7.3 External Revocation - Parallel Composition

This section presents the parallel composition of the remote-revocation components based on the described rule in 1. Consider the parallel composition $R_{rr} \parallel I_{rr}$. Program R_{rr} satisfies the assumption-commitment pair $(Ass^r_{rr},\ C^r_{rr})$ as follows:

$$\vdash \langle Ass^r_{rr},\ C^r_{rr} \rangle\ \{\varphi^r_{rr}\}\ R_{rr}\ \{\psi^r_{rr}\}$$

where the formal definition of φ^r_{rr}, ψ^r_{rr}, Ass^r_{rr}, and C^r_{rr} are as follows:

$$\begin{array}{lll}
\varphi^r_{rr} & \stackrel{\text{def}}{=} & \#C = \#D = 0 \ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{rr} & \stackrel{\text{def}}{=} & false \\
Ass^r_{rr} & \stackrel{\text{def}}{=} & \#C > 0 \rightarrow \\
 & & (isRrReq(last(B)) \ \wedge\ isUniquePID(last(B).spid) \wedge \\
 & & \ isValidInd(last(B).cap)) \\
C^r_{rr} & \stackrel{\text{def}}{=} & \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{array}$$

Program I_{rr} satisfies the assumption-commitment pair $(Ass^i_{rr},\ C^i_{rr})$ as follows:

$$\vdash \langle Ass^i_{rr},\ C^i_{rr} \rangle\ \{\varphi^i_{rr}\}\ I_{rr}\ \{\psi^i_{rr}\}$$

where the formal definition of φ^i_{rr}, ψ^i_{rr}, Ass^i_{rr}, and C^i_{rr} are as follows:

$$
\begin{array}{lcl}
\varphi^i_{rr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| \geqslant 1 \\
\psi^i_{rr} & \stackrel{\text{def}}{=} & false \\
Ass^i_{rr} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & \quad isRrReq(last(A)) \ \wedge \ isUniquePID(last(A).spid))) \\
 & & \quad isValidInd(last(A).cap)) \wedge \\
 & & (\#C = \#D > 0 \rightarrow isRrConfRes(last(D))) \\
C^i_{rr} & \stackrel{\text{def}}{=} & (\#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))) \wedge \\
 & & (\#C > 0 \rightarrow isRrConfRes(last(C)))
\end{array}
$$

By applying the parallel composition rule, we deduce as follows:

$$
\begin{array}{c}
\vdash \langle Ass_{rr}, \ C_{rr} \rangle \\
\{\varphi_{rr}\} \ R_{rr} \parallel I_{rr} \ \{\psi_{rr}\}
\end{array}
$$

where the formal definition of φ_{rr}, ψ_{rr}, Ass_{rr}, and C_{rr} are as follows:

$$
\begin{array}{lcl}
\varphi_{rr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0 \ \wedge \ |rcTree| \geqslant 1 \wedge \\
 & & |icTree| \geqslant 1 \\
\psi_{rr} & \stackrel{\text{def}}{=} & false \\
Ass_{rr} & \stackrel{\text{def}}{=} & \#A > 0 \rightarrow \\
 & & \quad (isRrReq(last(A)) \ \wedge \ isUniquePID(last(A).spid) \wedge \\
 & & \quad isValidInd(last(A).cap)) \\
C_{rr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D > 0 \rightarrow isRrConfRes(last(B))
\end{array}
$$

□

5.8 Fifth Lemma: Internal Revocation

The fifth lemma indicates that the local revocation of a valid p-cap by the delegator process will invalidate the delegated p-cap to another process inside the same node. Furthermore, the revocation operation encompasses all the re-delegated capabilities from the first delegated capability. Consequently, the resource controller participates in the local-revocation operation if one of the re-delegations includes a remote delegation. We formally define the lemma and proof it.

Lemma 5. *Given a process with a unique PID in a process node, an intermediate controller in the same node, and a resource controller inside a resource node, revoking a delegated p-cap to another process inside the same node invalidates the delegated p-cap and all its local and remote re-delegations.*

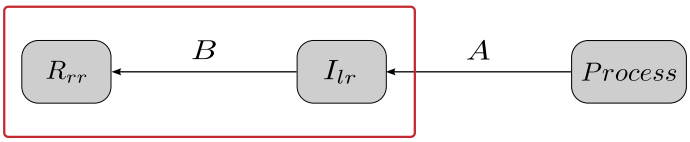


Figure 14: Syntactic Interfaces of the Internal-revocation Programs.

Proof. We use the assumption-commitment technique to prove the lemma.

When a process receives a locally delegated capability, it may perform a local or remote re-delegation. This re-delegation can include both local and remote delegations. Hence, the intermediate and resource controllers must participate in revoking the delegated capability and all the re-delegated capability in its sub-tree. To perform the revocation, program Ir collects the delegated capabilities in a list. Then, it iterates through the list and revokes the capabilities.

We model the intermediate controller at the delegating-process side and the resource controllers in a remote-revocation operation by programs $I_{rr} \stackrel{\text{def}}{=} (L^i_{lr}, T^i_{lr}, s^i_{lr}, t^i_{lr})$ and $R_{rr} \stackrel{\text{def}}{=} (L^r_{rr}, T^r_{rr}, s^r_{rr}, t^r_{rr})$, respectively. The programs and the process exchange messages via synchronous channels to handle local-revocation and possible remote-revocation requests. Figure 12 depicts these two programs and their syntactic interfaces. Program R_{rr} operates as explained in 5.7.1. Figures 15 and 13b illustrate sequential diagrams of programs I_{lr} and R_{rr}, respectively. In the next step, we prove that when the process sends a local-revocation request, programs I_{lr} and R_{rr} revoke the delegated capability along with all the re-delegated capabilities in its sub-tree.

Program I_{lr} receives a local capability-revocation request from program P_{lr} via channel A and removes the delegated capability and all its local or remote re-delegated capabilities from the capability tree with the participation of the resource controller. Finally, it returns revocation confirmation response to the process via channel B. The conditions and functions of I_{lr} are as follows:

$$
\begin{aligned}
init^i_{lr}(\sigma) &= (\sigma : rootPtr, isListEmpty \mapsto \\
&\qquad getRoot(\sigma(icTree), true)), \\
f_{i,1}(\sigma) &= (\sigma : capsList \mapsto \\
&\qquad getDelegatedCapsList(\sigma(rootPtr), \sigma(x_i.cap))), \\
f_{i,2}(\sigma) &= (\sigma : isListEmpty \mapsto IsCapsListEmpty(\sigma(capsList))), \\
f_{i,3}(\sigma) &= (\sigma : nextCap \mapsto getNextCap(\sigma(capsList)), \\
f_{i,4}(\sigma) &= (\sigma : isIndCap \mapsto isIndCapability(\sigma(nextCap))), \\
f_{i,5}(\sigma) &= (\sigma : revokeCap(\sigma(nextCap))), \\
f_{i,6}(\sigma) &= (\sigma : iIndCap \mapsto revokeInd(\sigma(nextCap))), \\
f_{i,7}(\sigma) &= (\sigma : y_i \mapsto genRrReq(\sigma(x_i), \sigma(iIndCap))), \\
f_{i,8}(\sigma) &= (\sigma : w_i \mapsto genRevConfRes(\sigma(z_i))))
\end{aligned}
$$

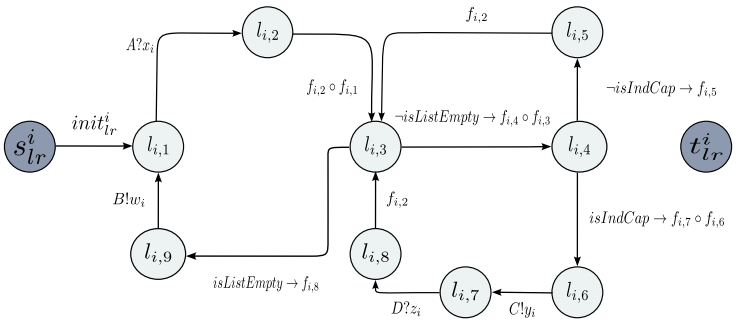


Figure 15: Internal Revocation - Program I_{lr}.

The A-C formula for program I_{lr} is as follows:

$$\vdash \langle Ass^i_{lr},\ C^i_{lr} \rangle\ \{\varphi^i_{lr}\}\ I_{lr}\ \{\psi^i_{lr}\}$$

The formal definition of φ^i_{lr}, ψ^i_{lr}, As^i_{lr}, and C^i_{lr} are as follows:

$$\begin{array}{lll}
\varphi^i_{lr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1 \\
\psi^i_{lr} & \stackrel{\text{def}}{=} & false \\
Ass^i_{lr} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & \quad isLrReq(last(A))\ \wedge\ isValidInd(last(A).cap))) \wedge \\
 & & (\#C = \#D > 0 \rightarrow isRrConfRes(last(D))) \\
C^i_{lr} & \stackrel{\text{def}}{=} & \#A = \#B > 0 \rightarrow isLrConfRes(last(B))
\end{array}$$

The assertion network for I_{lr} is as follows:

$$Q_{s^i_{lr}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D = 0 \ \wedge \ |icTree| \geqslant 1$$

$$Q_{l_{i,1}} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D \ \wedge \ isListEmpty$$

$$Q_{l_{i,2}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isLrReq(last(A)) \wedge$$
$$isUniquePID(last(A).spid) \ \wedge \ isValidInd(last(A).cap) \ \wedge \ isListEmpty$$

$$Q_{l_{i,3}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i$$

$$Q_{l_{i,4}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ \neg isListEmpty$$

$$Q_{l_{i,5}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ \neg isIndCap$$

$$Q_{l_{i,6}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \wedge$$
$$isIndCap \ \wedge \ isValidInd(iIndCap)$$
$$isRrReq(y_i) \ \wedge \ isUniquePID(y_i.spid) \ \wedge \ isValidInd(y_i.cap)$$

$$Q_{l_{i,7}} \stackrel{\text{def}}{=} \#B = \#D = (\#A - 1) = (\#C - 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i$$

$$Q_{l_{i,8}} \stackrel{\text{def}}{=} \#A = \#C = \#D = (\#B + 1) \ \wedge \ last(A) = x_i \ \wedge \ last(C) = y_i \ \wedge \ l$$
$$ast(D) = z_i \ \wedge \ isRrConfRes(z_i)$$

$$Q_{l_{i,9}} \stackrel{\text{def}}{=} \#B = \#C = \#D = (\#A - 1) \ \wedge \ last(A) = x_i \ \wedge \ isListEmpty \ \wedge \ isLrConfRes(z_i)$$

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption Ass^i_{lr}, commitment C^i_{lr}, and channels A, B, C, and D.

- $\models Q_{s^i_{lr}} \rightarrow C^i_{lr}$ follows from the above definitions.
- $\models Q_{s^i_{lr}} \wedge Ass^i_{lr} \rightarrow Q_{l_{i,1}} \circ init^i_{lr}$. In this internal transition, function $init^i_{lr}$ just assigns *true* to variable *isListEmpty*. Hence, this verification holds.
- $\models Q_{l_{i,1}} \ \wedge \ Ass^i_{lr} \rightarrow ((Ass^i_{lr} \rightarrow Q_{l_{i,2}}) \ \wedge \ C^i_{lr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : x_i, \ h \mapsto v, \ \sigma(h).(A, v))$ for arbitrary value v. In this input transition, just communication through channel A took place, and function g assigns the received value to x_i. Since $\#A > 0$, the first implication of Ass^i_{lr} satisfies $isLrReq(last(A))$, $isUniquePID(last(A).spid)$, and $isValidInd(last(A).cap)$. Thus, this verification holds.
- $\models Q_{l_{i,2}} \wedge Ass^i_{lr} \rightarrow Q_{l_{i,3}} \circ (f_{i,2} \circ f_{i,1})$. In this internal transition, function $f_{i,1}$ initialize the variable *capsList* with the delegated capability, which $last(A).cap$ points to it, and all the re-delegated capabilities in its sub-tree. Afterward, function $f_{i,2}$ checks if the list is empty. Thus, this verification holds.
- $\models Q_{l_{i,3}} \wedge \neg isListEmpty \wedge Ass^i_{lr} \rightarrow Q_{l_{i,4}} \circ (f_{i,4} \circ f_{i,3})$. In this internal transition, the condition implies that there is another capability in the list that the intermediate controller should revoke it. Function $f_{i,3}$ extracts this capability from the list, and

function $f_{i,4}$ checks if it is a remotely delegated capability. Hence, this verification holds.

- $\models Q_{l_{i,4}} \wedge \neg isIndCap \wedge Ass^i_{lr} \rightarrow Q_{l_{i,5}} \circ f_{i,5}$. The condition implies that the intermediate controller can revoke it locally in this internal transition. Hence, function $f_{i,5}$ revokes the capability locally by invalidating and removing it from the capability tree. Thus, this verification holds.
- $\models Q_{l_{i,5}} \wedge Ass^i_{lr} \rightarrow Q_{l_{i,3}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ checks if the list is empty. Hence, this verification holds.
- $\models Q_{l_{i,4}} \wedge isIndCap \wedge Ass^i_{lr} \rightarrow Q_{l_{i,6}} \circ (f_{i,7} \circ f_{i,6})$. The condition indicates that the intermediate and resource controller should collaborate because it is a remotely delegated capability. Hence, function $f_{i,6}$ copies the intermediate indicator capability, removes it from the tree, and assigns the copied intermediate indicator to variable $iIndCap$ when it returns. Afterward, function $f_{i,7}$ creates a capability-revocation request for program R_{rr} and assigns it to variable y_i, such that $isRevReq(y_i) = true$ and $y_i.spid = last(A).spid$. In addition, $\sigma' \models isUniquePID(y_i.spid)$ because it is equal to the receiving process id from the process, and Ass^i_{lr} implies that $isUniquePID(last(A).spid)$ Thus, this verification holds.
- $\models Q_{l_{i,6}} \wedge Ass^i_{lr} \rightarrow ((Ass^i_{lr} \rightarrow Q_{l_{i,7}}) \wedge C^i_{lr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(C, \sigma(y_i)))$. In this output transition, program I_{lr} sends y_i through channel C. C^i_{lr} holds because $last(C) = \sigma(y_i)$, and from $Q_{l_{i,7}}$, we have $isRevReq(y_i)$, $isUniquePID(y_i.spid)$, and $isValidInd(y_i.cap)$. Hence, this verification holds.
- $\models Q_{l_{i,7}} \wedge Ass^i_{lr} \rightarrow ((Ass^i_{lr} \rightarrow Q_{l_{i,8}}) \wedge C^i_{lr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : z_i,\ h \mapsto v,\ \sigma(h).(D, v))$ for arbitrary value v. In this input transition, a communication through channel D took place, and function g assigns the received value to z_i. Since $\#D > 0$, Ass^i_{lr} implies that $\sigma' \models isRrConfRes(z_i)$. Thus, this verification holds.
- $\models Q_{l_{i,8}} \wedge Ass^i_{lr} \rightarrow Q_{l_{i,3}} \circ f_{i,2}$. In this internal transition, function $f_{i,2}$ checks if the list is empty. Hence, this verification holds.
- $\models Q_{l_{i,3}} \wedge isListEmpty \wedge Ass^i_{lr} \rightarrow Q_{l_{i,9}} \circ (f_{i,8} \circ unlockTree)$. In this internal transition, the condition implies that the intermediate controller has revoked all the delegated capabilities from the list. Afterward, function $f_{i,8}$ creates a revocation-confirmation response for the process and assigns it to w_i such that $isLrConfRes(w_i)$. Thus, this verification holds.
- $\models Q_{l_{i,9}} \wedge Ass^i_{lr} \rightarrow ((Ass^i_{lr} \rightarrow Q_{l_{i,1}}) \wedge C^i_{lr}) \circ g$, where $g(\sigma) \stackrel{\text{def}}{=} (\sigma : h \mapsto \sigma(h).(B, \sigma(w_i)))$. In this output transition, program I_{lr} sends w_i through channel B. Commitment C^i_{lr} holds because $\sigma \models isLrConfRes(w_i)$. Hence, this verification holds.

Since $Q_{t^i_{lr}} \rightarrow \psi^i_{lr}$, the post-condition of program I_{lr} is satisfied.

5.8.1 Internal Revocation - Parallel Composition

This section presents the parallel composition of the remote-revocation components based on the described rule in 1. Consider the parallel composition $R_{rr} \parallel I_{lr}$. Program

R_{rr} satisfies the assumption-commitment pair $(Ass^r_{rr},\ C^r_{rr})$ as follows:

$$\vdash \langle Ass^r_{rr},\ C^r_{rr}\rangle\ \{\varphi^r_{rr}\}\ R_{rr}\ \{\psi^r_{rr}\}$$

where the formal definition of φ^r_{rr}, ψ^r_{rr}, Ass^r_{rr}, and C^r_{rr} are as follows:

$$\begin{array}{lll}
\varphi^r_{rr} & \stackrel{\text{def}}{=} & \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1 \\
\psi^r_{rr} & \stackrel{\text{def}}{=} & false \\
Ass^r_{rr} & \stackrel{\text{def}}{=} & \#C > 0 \rightarrow \\
 & & (isRrReq(last(B))\ \wedge\ isUniquePID(last(B).spid)\ \wedge \\
 & & \ isValidInd(last(B).cap)) \\
C^r_{rr} & \stackrel{\text{def}}{=} & \#C = \#D > 0 \rightarrow isRrConfRes(last(D))
\end{array}$$

Program I_{rr} satisfies the assumption-commitment pair $(Ass^i_{lr},\ C^i_{lr})$ as follows:

$$\vdash \langle Ass^i_{lr},\ C^i_{lr}\rangle\ \{\varphi^i_{lr}\}\ I_{lr}\ \{\psi^i_{lr}\}$$

The formal definition of φ^i_{lr}, ψ^i_{lr}, As^i_{lr}, and C^i_{lr} are as follows:

$$\begin{array}{lll}
\varphi^i_{lr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0\ \wedge\ |icTree| \geqslant 1 \\
\psi^i_{lr} & \stackrel{\text{def}}{=} & false \\
Ass^i_{lr} & \stackrel{\text{def}}{=} & (\#A > 0 \rightarrow \\
 & & \ isLrReq(last(A))\ \wedge\ isValidInd(last(A).cap))) \wedge \\
 & & (\#C = \#D > 0 \rightarrow isRrConfRes(last(D))) \\
C^i_{lr} & \stackrel{\text{def}}{=} & \#A = \#B > 0 \rightarrow isLrConfRes(last(B))
\end{array}$$

By applying the parallel composition rule, we deduce as follows:

$$\begin{array}{c}
\vdash \langle Ass_{lr},\ C_{lr}\rangle \\
\{\varphi_{lr}\}\ R_{lr}\ \|\ I_{lr}\ \{\psi_{lr}\}
\end{array}$$

where the formal definition of φ_{lr}, ψ_{lr}, Ass_{lr}, and C_{lr} are as follows:

$$\begin{array}{lll}
\varphi_{lr} & \stackrel{\text{def}}{=} & \#A = \#B = \#C = \#D = 0\ \wedge\ |rcTree| \geqslant 1\ \wedge \\
 & & \ |icTree| \geqslant 1 \\
\psi_{lr} & \stackrel{\text{def}}{=} & false \\
Ass_{lr} & \stackrel{\text{def}}{=} & \#A > 0 \rightarrow \\
 & & (isLrReq(last(A))\ \wedge\ isValidInd(last(A).cap)) \\
C_{lr} & \stackrel{\text{def}}{=} & (\#A = \#B > 0 \wedge \#C = \#D > 0) \rightarrow \\
 & & \ isLrConfRes(last(B))
\end{array}$$

□

5.8.2 Capability Safety

Capability safety implies that a process can only obtain its capabilities through a resource allocation or a capability delegation and only employ its valid capabilities.

Definition 1. *Capability System. A capability system is a tuple comprising the following:*

- *a capability set* $\mathbb{C} = \{\mathbb{C}_r \cup \mathbb{C}_i \cup \mathbb{C}_p\}$;
- *a function:* $rsrc : \mathbb{C} \rightarrow \mathbb{R}$
- *a function:* $priv : \mathbb{C} \rightarrow 2^{\mathbb{V}}$
- *a function:* $pCaps : \mathbb{P} \rightarrow 2^{\mathbb{C}_p}$
- *a function:* $cAuth : \mathbb{C} \rightarrow \mathbb{R} \times 2^{\mathbb{V}}$
- *a function:* $pAuth : \mathbb{P} \rightarrow 2^{\mathbb{R}} \times 2^{\mathbb{V}}$

The tuple must satisfy the following conditions to denote a valid capability system:
$\forall\ c_p \in \mathbb{C}_p, c_i \in \mathbb{C}_i, c_r \in \mathbb{C}_r, T_i \in \mathbb{T}_i,$ *and* $T_r \in \mathbb{T}_r$

(a) $c_p \mapsto c_i \Rightarrow cAuth(c_p) \subseteq cAuth(c_i)$

(b) $c_i \mapsto c_r \Rightarrow cAuth(c_i) \subseteq cAuth(c_r)$

(c) $isInTree(T_i, c_i, c_i') \Rightarrow cAuth(c_i') \subseteq cAuth(c)$ *s.t.* $c_i \neq c_i^{root}$

(d) $isInTree(T_r, c_r, c_r') \Rightarrow cAuth(c_r') \subseteq cAuth(c_r)$

The valid-capability set of a process indicates the legitimately acquired capabilities by the process that are still valid; thus, the process can employ them for read/write operations. We formally define this set as follows:

$$validCaps(p) \subseteq pCaps(p) : (\forall c_p \in pCaps(p) \ \ s.t. \\ isValidProCap(c_p) \Rightarrow c_p \in validCaps(p))$$

Now, we define the capability-safety property and prove that our capability-based access-control system is capability safe.

Definition 2. *Capability Safety. An access-control system is capability safe, concerning the capability system* $(\mathbb{C}, rsrc, priv, pCap, cAuth, pAuth)$*, if the following condition holds for all processes* $p \in \mathbb{P}$:

- $rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$

Where $rsrcSet$ indicates accessible resources by the process.

Proof. The proof proceeds by induction on the structure of the process's capability set. The proof consists of one base case and two inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the process possesses no valid capability; thus, the process can access no resources.
- **Inductive Case One:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 1 to 3, the process receives a valid process capability, c_p^{new}, due to a resource-allocation or internal/external-delegation request. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \cup c_p^{new}$$

  Hence, its new resource list transforms as follows:

  $$rsrcSet'(p) = rsrcSet(p) \cup rsrc(c_p^{new})$$

- **Inductive Case Two:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 4 to 5, one of the process's capabilities, c_p^{old}, becomes invalid due to an internal/external revocation. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \setminus c_p^{old}$$

  Thus, its new resource list transforms as follows:

  $$rsrcSet'(p) = rsrcSet(p) \setminus rsrc(c_p^{old})$$

□

5.8.3 Authority Safety

We claim that a distributed capability-based access-control system is authority safe if it addresses the following properties regarding the object-capability model:

- **An authority's owner obtained it through a capability delegation**: A process can acquire authority only if the owner of the authority delegates it to the process.
- **No authority amplification**: The capability owners can only delegate what authority they have.

Our access-control system guarantees both properties. Processes can only obtain capabilities through allocating resources or delegating capabilities by other processes. Furthermore, using well-formed capability trees, the access-control system ensures that no process can delegate an authority it does not own. We define the authority-safety property as follows:

Definition 3. *Authority Safety. A capability-based access-control system is authority safe, concerning the capability system* ($\mathbb{C}$, $rsrc$, $priv$, $pCap$, $cAuth$, $pAuth$)*, if the following condition holds for all processes* $p \in \mathbb{P}$:[1]

- $$pAuth(p) = \bigcup_{c_p \in validCaps(p)} cAuth(c_p)$$

Proof. The proof proceeds by induction on the structure of the process's capability set. The proof consists of one base case and two inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the process possesses no valid capability; thus, the process can access no resources.
- **Inductive Case One:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 1 to 3, the process receives a valid process capability, c_p^{new}, due to a resource-allocation or internal/external-delegation request. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \cup c_p^{new}$$

  Hence, its authority transforms as follows:

  $$pAuth'(p) = pAuth(p) \cup cAuth(c_p^{new})$$

  In addition, because all the capability trees are well-formed, the authority of the delegated capability is always a subset of the orginal capability.
- **Inductive Case Two:** $validCaps(p)$ contains the current process's valid capabilities; hence, we have:

  $$rsrcSet(p) = \bigcup_{c_p \in validCaps(p)} rsrc(c_p)$$

  Based on Lemmas 4 to 5, one of the process's capabilities, c_p^{old}, becomes invalid due to an internal/external revocation. Its valid-capability set changes as follows:

  $$validCaps'(p) = validCaps(p) \setminus c_p^{old}$$

  Thus, its authority transforms as follows:

  $$pAuth'(p) = pAuth(p) \setminus pAuth(c_p^{old})$$

□

5.9 Isolation Property

The isolation property expresses that two processes cannot access to the same resource. This property guarantees that untrusted processes cannot access trusted processes' resources. We formally define this property as follows:

Definition 4. *Isolation Property. Given a set of Resources* $R \subseteq \mathbb{R}$ *and a set of processes* $p_1, \cdots, p_k \in P \subseteq \mathbb{P}$*, we have* $Isolation(R, p_1, \cdots, p_k \in P)$ *if*:

- $\forall\, i, j\colon (1 \leqslant i < j \leqslant k \;\wedge\; rsrcSet(p_i) \subseteq R \;\wedge\; rsrcSet(p_i) \subseteq R)$
  $\Rightarrow\; rsrcSet(p_i) \cap rsrcSet(p_j) = \emptyset$

Processes can isolate their resources by delegating no capability. However, this approach is error-prone because it relies on developers to ensure processes never delegate their capabilities. Therefore, we support the isolation automatically using new permission. Let $\mathbb{Y} = \{d\}$ be the capability-related permission set, where d denotes the delegation permission. When controllers generate capabilities due to allocating resources, they unset this permission. Thus, because processes cannot delegate their capabilities and our capability-based access-control system is capability and authority safe, it supports the isolation property.

Proof. The proof proceeds by induction on the structure of capability trees.

we start with the resource capability. The proof consists of one base case and three inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the resource capability tree only contain the root capability; thus, it satisfies the isolation property.
- **Inductive Case One:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a resource-allocation request. Based on 1–**??** the resource controller allocates the resource which is free and adds its corresponding resource capability to tree. Furthermore, it unset the delegation permission, d, in the resource capability. Because the new capability does not have any overlap with the existing ones, the tree satisfies the isolation property. Moreover, it creates the corresponding intermediate capability, in which the delegation permission is unset.
- **Inductive Case Two:**The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a external-delegation request. Because the delegation permission is unset in all the resource capabilities inside the tree, the resource controller rejects the request and does not change the tree. Thus, the tree satisfies the isolation property.
- **Inductive Case Three:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a external-revocation request. However, because the delegation permission is unset in all the capabilities inside the tree, none of them have a delegation-hierarchy as its sub-tree. Thus, the resource controller reject the request because the intermediate capability inside the request points to an non-existing resource capability. Therefore, the tree does not change and it satisfies the isolation property.

Now, we proof the isolation property for the intermediate tree. The proof consists of one base case and three inductive cases.

- **Base Case:** The capability set is empty.
  In this case, the intermediate capability tree only contain the root capability; thus, it satisfies the isolation property.
- **Inductive Case One:** The capability contains more than one capability and it satisfies the isolation property. The intermediate controller receives a resource-allocation response and inserts it as the direct child of the tree's root. Because the resource in the received intermediate capability does not have overlap with any resources in the resource node, it does not have overlap with any capability in the resource node. Thus, it does not have any overlap with the existing intermediate capability inside the intermediate tree. Thus, the tree satisfies the isolation property.
- **Inductive Case Two:**The capability contains more than one capability and it satisfies the isolation property. The intermediate controller receives a internal-delegation request. Because the delegation permission is unset in all the intermediate capabilities inside the tree, the intermediate controller rejects the request and does not change the tree. Thus, the tree satisfies the isolation property.
- **Inductive Case Three:** The capability contains more than one capability and it satisfies the isolation property. The resource controller receives a internal-delegation request. However, because the delegation permission is unset in all the capabilities inside the intermediate tree, none of them have a delegation-hierarchy as its sub-tree. Thus, the resource controller reject the request because the process capability inside the request points to an non-existing intermediate capability. Hence, the tree does not change and it satisfies the isolation property.

Therefore, we proved that our access-control system supports the isolation property. □

6 Use Case

MDC [36] provides direct access to the shared memory pool. The direct access to the shared memory introduces a vulnerability to the MDC's architecture. For example, imagine a process that uses the shared memory pool as the communication medium due to its direct and memory-speed persistence access; the process writes data on the shared memory and passes the data's reference to another process to communicate with it. However, a third process can read the communicated data by knowing the reference and violate the security or privacy of data. Therefore, MDC requires an access-control system to preserve the security and privacy of data in the shared memory pool.

We instantiates our capability-based access-control system for MDC as the use case and demonstrate how it handles requests and capabilities[1]. In our general-purpose system, programs directly connect to other programs via synchronous channels (as depicted in Figure 6). However, in MDC, they are part of the instantiated controllers. An instantiated controller integrates all the related components inside the controller and connects them to the rest of the system via *handler*s and bridges. In addition, we add a module to OS which connects processes to the intermediate controller. We assume adversaries cannot compromise this module. This section describes describes nodes and controllers in the instantiated system, their components, and their syntactic interfaces. Furthermore, we explain how they cooperate to control access.

The new components change the environments of the existing components. In addition, new components commit to guaranteeing some of the previous assumptions. For example, the OS module checks the process ID in a request against the sender's PID because processes may act maliciously. Thus, it commits that each request contains a unique process ID, and it belongs to the sender. To prove the soundness of the instantiated system, we verify that it guarantees security properties. Thus, because the proofs of the properties depend on Lemmas 1 to 5, we must refine lemmas' proofs.

To refine Lemmas' proofs, we first verify the new modules regarding the assumption-commitment method. Afterward, we instantiate controllers by integrating the verified modules and applying the parallel composition rule (Rule 1). The soundness of the parallel composition rule depends on the validity of the A-C formulas of the integrated modules and the validity of the hypothesis of Rule 1 ($A \wedge C_1 \rightarrow A_2$, $A \wedge C_2 \rightarrow A_1$) for the joint channels between modules [64]. Finally, we verify the lemmas; thus, we prove that the instantiated system guarantees the security properties. Section *Use Case* of the Technical Report represents the whole proof.

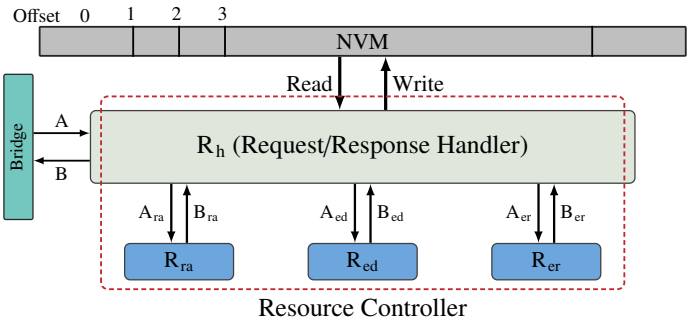
6.1 Resource Node

Figure 16a depicts a resource node in the instantiated system, including a resource controller and a byte-addressable NVM.

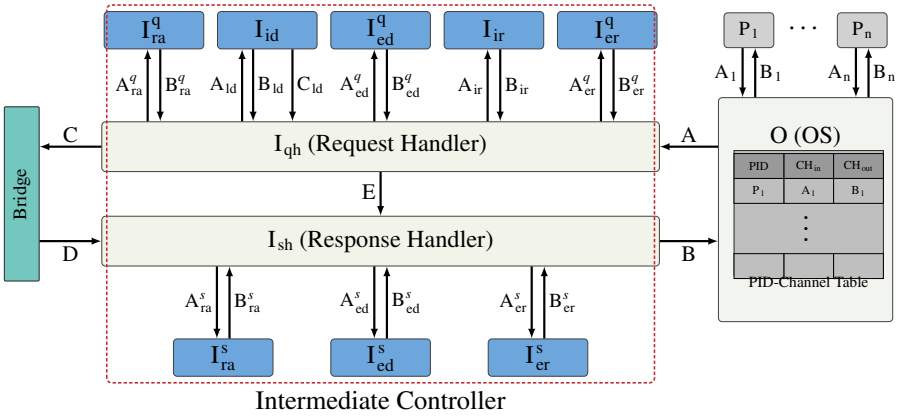
6.1.1 Resource Controller

The instantiated resource controller comprises four components: Request/Response handler (program R_h), Resource Allocation (R_{ra}), External Delegation (R_{ed}), and Ex-

[1] **??** contains the capability structures of the instantiated system.



(a) Resource Node's Components and their Syntactic Interfaces.



(b) PE Node's Components and their Syntactic Interfaces.

Figure 16: Instantiated Access-Control System for MDC.

ternal Revocation (R_{er}). Each component has its assumptions and commitments. We define program R as the parallel execution of the programs in the controller as follows:

$$R \stackrel{\text{def}}{=} R_h \parallel R_{ra} \parallel R_{ed} \parallel R_{er}$$

With the A-C formula $\vdash \langle A^r, C^r \rangle : \{\varphi^r\} R \{\psi^r\}$. By applying the parallel composition rule, the commitments of the controller should be the commitments of all its components ($C_h \wedge C_{ra} \wedge C_{ed} \wedge C_{er}$). Furthermore, we should check the correctness of the controller's assumptions by checking the validity of the hypothesis ($A \wedge C_1 \rightarrow A_2$, $A \wedge C_2 \rightarrow A_1$) in Rule 1. It means, the controller's assumption and the handler's commitments should guarantee the assumption of programs R_{ra} (A_{ra}), R_{ed} (A_{ed}), and R_{er} (A_{er}).

Program R_h receives requests from channel A and processes them in a *First In / First Out* (FIFO) order. It checks each request and commits to forwarding the request to the corresponding program. Program R_h assumes that each received request contain a unique process ID in the sender's PE node. Because each request contains a unique process ID and program R_h commits to forwarding it to the right program, which sat-

isfies the programs' assumptions ($A^r \wedge C_h \rightarrow A_{ra,ed,er}$). Furthermore, these programs commit to returning valid responses (C_{ra}, C_{ed}, C_{er}). Thus, combining these commitments and assumption of channel A ($A^r \wedge C_{ra,ed,er}$) guarantees that R_h will receive valid responses, which it will send out through channel D. Therefore, the resource controller commits to return valid responses via channel D when it receives requests containing unique process IDs from channel A.

6.2 PE Node

Figure 16b illustrates a PE node in the instantiated system, comprising an intermediate controller, an operating system (OS), and running processes in the node.

6.2.1 Operating System

OS acts as a mediator between processes and the intermediate controller. We model the operating system in the PE node by program O. It has two tasks: assigning unique ID/communication channels to each process and forwarding received messages to appropriate channels. Program O employs a table (*PID-Channel*) to map process IDs to in/out channels (Figure 16b). Therefore, it can check if the process ID in a request belongs to the sender. Furthermore, O forwards received responses to processes.

Program O commits to the intermediate controller that the PID in a request belongs to the sender and is unique inside the node by performing the first task and checking requests against the table. Furthermore, it commits to processes that they will receive their valid responses using the table. We denote the A-C formula of program O with $\vdash \langle A^o,\ C^o \rangle :\ \{\varphi^o\}\ O\ \{\psi^o\}$.

6.2.2 Intermediate Controller

We follow the pattern in which we reasoned about the resource controller to reason about the intermediate controller regarding the parallel composition rule. We model the intermediate controller by program I as the parallel execution of the above programs as follows:

$$I \stackrel{\text{def}}{=} I_{qh} \parallel I^q_{ra} \parallel I_{id} \parallel I^q_{ed} \parallel I_{ir} \parallel I^q_{er} \parallel I_{sh} \parallel I^s_{ra} \parallel I^s_{ed} \parallel I^s_{er}$$

With the A-C formula $\vdash \langle A^i, C^i \rangle :\ \ \{\varphi^i\}\ I\ \{\psi^i\}$, after applying the parallel composition rule. The commitments of the intermediate controller will be the combination of its programs. Assumption A^i indicates that the intermediate controller assumes the process ID inside each request is unique in the PE node, and responses from resource controllers contain valid capabilities. However, before checking the correctness of its assumptions, we describe how they cooperate to handle requests and responses.

The intermediate controller handles requests and responses in parallel via request and response handlers Each handler has three tasks: locking the capability tree, forwarding its input to the appropriate module, and sending the response of a module to the output channel. Parallel handling of requests and responses may cause manipulating the tree simultaneously, thus,breaking the tree's well-formedness The handlers employ a locking mechanism ($\mathscr{L}$) to lock the tree before processing each input to prevent

the situation. We assume $\mathscr{L}$ performs the locking task correctly and fairly. Therefore, controllers can always guarantee the tree's well-formedness. We now explain how each handler accomplishes its task.

Request Handling. We model the request handler by the program I_{qh}. Program I_{qh} cooperates with five other programs to handle requests, including I^q_{ra} (resource-allocation request handler), I_{id} (the internal-delegation handler), I^q_{ed} (the external-delegation request handler), I_{ir} (the internal-revocation handler), and I^q_{er} (the external-revocation request handler). Program I_{qh} assumes that the PID in a request belongs to the sender and is unique inside the node. I_{qh} commits to forward the request to the corresponding program and forwarding programs' responses via output channels.

Response Handling. We model the response handler by the program I_{sh}. It cooperates with three programs to handle responses, including I^s_{ra} (resource-allocation response handler), I^s_{ed} (the external-delegation response handler), and I^s_{er} (the external-revocation response handler). In addition, I_{sh} directly forwards all the responses from the request handler to O. Program I_{sh} assumes that all received responses from the resource controllers are valid. Furthermore, it commits to forwarding received responses to the corresponding programs and forwarding programs' responses to O directly.

Now, we check the validity of the hypothesis in Rule 1. Program I_{qh} receives requests from channel A and processes them in a FIFO order. Because I_{qh} assumes each request contains a unique process ID and commits to forwarding the request to the right program, it satisfies the assumption of programs I^q_{ra} (A^q_{ra}), I_{id} (A_{id}), I^q_{ed} (A^q_{ed}), I_{ir} (A_{ir}), and I^q_{er} (A^q_{er}). I_{sh} processes received responses from channel D. Because each response contains a valid capability and I_{sh} commits forwarding it to the right program. Furthermore, I_{sh} receives valid responses from programs I^s_{ra}, I^s_{ed}, and I^s_{er}, and forward them via channel B.

6.3 Parallel Composition

The instantiated access-control system (denoted by program S) comprises programs R, I, and O, running in parallel ($S \stackrel{\text{def}}{=} R \parallel I \parallel O$). Hence, We apply the parallel composition rule to acquire the instantiated system's A-C formula as the following:

$$\vdash \langle A^s,\, C^s \rangle :\ \{\varphi^s\}\ S\ \{\psi^s\}$$

Program O commits to program I that each process uses its PID when sending a request, and the PID is unique. Furthermore, program O commits to forwarding responses to their corresponding processes directly. In addition, the commitments of programs R and I comprise their modules' commitments. Therefore, the above A-C formula proves Lemmas 1 to 5 for the instantiated system.

References

[1] K. Keeton, The machine: An architecture for memory-centric computing, in *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, volume 10, 2015.

[2] K. Asanović, FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, In FAST (2014).

[3] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, et al., Enzian: an open, general, CPU/FPGA platform for systems software research, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 434–451, 2022.

[4] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, et al., Understanding {Rack-Scale} Disaggregated Storage, in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[5] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs, in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al., A reconfigurable fabric for accelerating large-scale datacenter services, ACM SIGARCH Computer Architecture News **42**(3), 13–24 (2014).

[7] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, System-level implications of disaggregated memory, in *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, IEEE, 2012.

[8] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, Network requirements for resource disaggregation, in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.

[9] S. Angel, M. Nanavati, and S. Sen, Disaggregation and the Application, in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[10] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, Can far memory improve job throughput?, in *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[11] D. Gouk, S. Lee, M. Kwon, and M. Jung, Direct access, High-Performance memory disaggregation with DirectCXL, in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[12] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, Clio: A hardware-software co-designed disaggregated memory system, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.

[13] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, Mind: In-network memory management for disaggregated data centers, in *Proceedings*

of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 488–504, 2021.

[14] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, Intel® omni-path architecture: Enabling scalable, high performance fabrics, in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, IEEE, 2015.

[15] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, Efficient memory disaggregation with infiniswap, in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, pfabric: Minimal near-optimal datacenter transport, ACM SIGCOMM Computer Communication Review **43**(4), 435–446 (2013).

[17] C. E. L. (CXL), CXL Specification, `https://www.computeexpresslink.org/download-the-specification`, 2023 (accessed April 16, 2023).

[18] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, et al., Rack-scale disaggregated cloud data centers: The dReDBox project vision, in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 690–695, IEEE, 2016.

[19] A. D. Papaioannou, R. Nejabati, and D. Simeonidou, The benefits of a disaggregated data centre: A resource allocation approach, in *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, IEEE, 2016.

[20] R. Lin, Y. Cheng, M. De Andrade, L. Wosinska, and J. Chen, Disaggregated data centers: Challenges and trade-offs, IEEE Communications Magazine **58**(2), 20–26 (2020).

[21] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, Disaggregated memory for expansion and sharing in blade servers, ACM SIGARCH computer architecture news **37**(3), 267–278 (2009).

[22] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, Shoal: A Network Architecture for Disaggregated Racks., in *NSDI*, pages 255–270, 2019.

[23] M. Nanavati, J. Wires, and A. Warfield, Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage., in *NSDI, Volume 17*, pages 17–33, 2017.

[24] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation, in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880, IEEE, 2020.

[25] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, et al., Pond: CXL-based memory pooling systems for cloud platforms, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[26] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, et al., Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms, IEEE Micro **43**(2), 30–38 (2023).

[27] B. Caldwell, S. Goodarzy, S. Ha, R. Han, E. Keller, E. Rozner, and Y. Im, Fluidmem: Full, flexible, and fast memory disaggregation for the cloud, in *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 665–677, 2020.

[28] J. Wahlgren, M. Gokhale, and I. B. Peng, Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems, arXiv preprint arXiv:2211.02682 (2022).

[29] V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, Deact: Architecture-aware virtual memory support for fabric attached memory systems, in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 453–466, IEEE, 2021.

[30] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, FaRM: Fast remote memory, in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[31] D. Minturn, Nvm express over fabrics, in *11th Annual OpenFabrics International OFS Developers' Workshop*, 2015.

[32] S.-Y. Tsai and Y. Zhang, A double-edged sword: security threats and opportunities in one-sided network communication, in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, pages 3–3, 2019.

[33] N. R. Herbst, S. Kounev, and R. H. Reussner, Elasticity in Cloud Computing: What It Is, and What It Is Not., in *ICAC*, volume 13, pages 23–27, 2013.

[34] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park, Elastic resource sharing for distributed deep learning, in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[35] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, Semeru: A memory-disaggregated managed runtime, in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 261–280, 2020.

[36] K. Keeton, Memory-Driven Computing., in *FAST*, 2017.

[37] J. B. Dennis and E. C. Van Horn, Programming semantics for multiprogrammed computations, Communications of the ACM **9**(3), 143–155 (1966).

[38] R. S. Fabry, Capability-based addressing, Communications of the ACM **17**(7), 403–412 (1974).

[39] N. P. Carter, S. W. Keckler, and W. J. Dally, Hardware support for fast capability-based addressing, ACM SIGOPS Operating Systems Review **28**(5), 319–327 (1994).

[40] K. M. Bresniker, P. Faraboschi, A. Mendelson, D. Milojicic, T. Roscoe, and R. N. Watson, Rack-scale capabilities: fine-grained protection for large-scale memories, Computer **52**(2), 52–62 (2019).

[41] M. Hille, N. Asmussen, P. Bhatotia, and H. Härtig, Semperos: A distributed capability system, in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 709–722, 2019.

[42] J. H. Saltzer, Protection and the control of information sharing in Multics, Communications of the ACM **17**(7), 388–402 (1974).

[43] R. M. Needham and R. D. Walker, The Cambridge CAP computer and its protection system, ACM SIGOPS Operating Systems Review **11**(5), 1–10 (1977).

[44] A. K. Jones, R. J. Chansler Jr, I. Durham, K. Schwans, and S. R. Vegdahl, StarOS, a multiprocessor operating system for the support of task forces, in *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 117–127, 1979.

[45] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, IBM System/38 support for capability-based addressing, in *Proceedings of the 8th annual symposium on Computer Architecture*, pages 341–348, 1981.

[46] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, Hardbound: architectural support for spatial safety of the C programming language, ACM SIGOPS Operating Systems Review **42**(2), 103–114 (2008).

[47] G. W. Cox, W. M. Corwin, K. K. Lai, and F. J. Pollack, A unified model and implementation for interprocess communication in a multiprocessor environment, in *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 125–126, 1981.

[48] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, The CHERI capability model: Revisiting RISC in an age of risk, in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, IEEE, 2014.

[49] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, Hydra: The kernel of a multiprocessor operating system, Communications of the ACM **17**(6), 337–345 (1974).

[50] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, et al., Overview of the Chorus distributed operating system, in *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[51] S. J. Mullender, G. Van Rossum, A. Tananbaum, R. Van Renesse, and H. Van Staveren, Amoeba: A distributed operating system for the 1990s, Computer **23**(5), 44–53 (1990).

[52] R. F. Rashid and G. G. Robertson, Accent: A communication oriented network operating system kernel, ACM SIGOPS Operating Systems Review **15**(5), 64–75 (1981).

[53] N. Hardy, KeyKOS architecture, ACM SIGOPS Operating Systems Review **19**(4), 8–25 (1985).

[54] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, Mach: A new kernel foundation for UNIX development, (1986).

[55] J. S. Shapiro, J. M. Smith, and D. J. Farber, EROS: a fast capability system, in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[56] J. Liedtke, On micro-kernel construction, ACM SIGOPS Operating Systems Review **29**(5), 237–250 (1995).

[57] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, The multikernel: a new OS architecture for scalable multicore systems, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[58] L. Azriel, L. Humbel, R. Achermann, A. Richardson, M. Hoffmann, A. Mendelson, T. Roscoe, R. N. Watson, P. Faraboschi, and D. Milojicic, Memory-side protection with a capability enforcement co-processor, ACM Transactions on Architecture and Code Optimization (TACO) **16**(1), 1–26 (2019).

[59] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, Grid information services for distributed resource sharing, in *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, pages 181–194, IEEE, 2001.

[60] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, Themis: Fair and efficient GPU cluster scheduling, in *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[61] G. J. Holzmann, The model checker SPIN, IEEE Transactions on software engineering **23**(5), 279–295 (1997).

[62] M. S. Miller, K.-P. Yee, J. Shapiro, et al., Capability myths demolished, Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research . . . , 2003.

[63] J. Misra and K. M. Chandy, Proofs of networks of processes, IEEE transactions on software engineering (4), 417–426 (1981).

[64] W.-P. De Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, *Concurrency verification: Introduction to compositional and noncompositional methods*, volume 54, Cambridge University Press, 2001.