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

Anonymous Author(s)

ABSTRACT

Disaggregated-resource architectures address growing resource and scalability demands from distributed computing systems in the big-data era. However, the distributed nature of shared resources in disaggregated-resource architectures makes controlling access to resources challenging. Recent works showed that access control based on capabilities suits these architectures better. In this paper, we reviewed the existing capability systems and extracted a minimum set of criteria that an access control system should satisfy to suit a distributed architecture. We concluded that none of the existing approaches meet all these criteria. Therefore to address their shortcomings, we propose a general-purpose capability-based access-control system, SADRA. We formally prove that SADRA's approach is sound; in particular, we prove that it is capability and authority safe. In addition, we show that SADRA can provably isolate resources of processes, thus protecting them from illegal accesses. To show that disaggregated-resource architecture can benefit from our access control system, we apply SADRA to Memory-Driven Computing as a case study. Finally, we mechanized our approach's soundness by modeling and verifying it using the SPIN model checker.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Access control**; **Formal security models**.

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 [5, 15, 39, 44, 53, 62]. Driven by the growing resource demands, disaggregated-resource architectures have recently attracted considerable attention from academia and industry [2–4, 10, 17, 25, 26, 28, 29, 43, 48]. For example, various architectures and applications, such as *data centers* [38, 49, 60], *blade servers* [47], *rack-scale systems* [57, 61, 68], *cloud frameworks* [9, 12, 45], and *high-performance computing* [42, 72], 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 [10, 22, 39, 54]. These controllers receive requests and directly read/write from/to resources based on them, thus raising security and privacy concerns [71].

Supporting elasticity introduces a new challenge to access-control systems. Elasticity in disaggregated-resource architectures means the ability and flexibility to scale [31, 35, 73]—resources and PEs should be able to attach to an architecture with minimum adaptation. For example, consider *Memory-Driven Computing* (MDC) [40], 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 [13, 20, 23] best suits disaggregated-resource architectures [11, 32]. 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 [11]. Moreover, it supports the *least-privilege principle* [66], 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 [1, 7, 8, 13, 16, 21, 30, 32, 34, 36, 46, 56, 58, 63, 65, 67, 74, 75]. We systematically reviewed existing capability systems to investigate which suit the architectures best. During our reviewing capability systems, we extracted six traits from them, including distributed structure, subject and object granularity, capability delegation and revocation, and persistency (Section 3). 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* [74], 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* [7] and *SemperOS* [32], 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

CCS, November 26–30, 2023, Copenhagen, Denmark
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

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* [18] and *GPU clustering* [51], 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 accessible resources by processes are mutually exclusive.

We instantiate our access-control system for MDC as the use case (Section 6). 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 and check our design using the *SPIN* model checker [33] (Section 7). Specifically, we employ SPIN to check the safety properties based on the assumption-commitment method.

The contributions of our work are as follows[1]:

- 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* [20, 23]. Miller et al. [52] 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 [19, 55] 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

[1]The Technical Report and SPIN models are accessible at https://sadra-acs.github.io

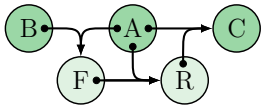


Figure 1: Capability-Object Model.
Legend: F: Forwarding Facet; R: Revoking Facet

process with the following A-C correctness formula:

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

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 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,* $p \overset{def}{=} p_1 \parallel p_2$*, is as follows:*

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

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

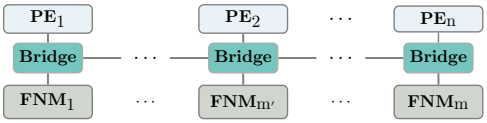


Figure 2: MDC Architecture.
Legend: FNM: Fabric-attached NVM Module

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. [11] 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.

Table 1: Capability-based Access-control Systems

	CAP [58]	StarOS [36]	IBM/38 [34]	Hardbound [21]	iAPX432 [16]	CHERI [74]	M-Machine [13]	Hydra [75]	Chorus [65]	Amoeba [56]	Accent [63]	KeyKOS [30]	Mach [1]	EROS [67]	L4 [46]	Barrelfish [8]	CEP [7]	SemperOS [32]	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

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.

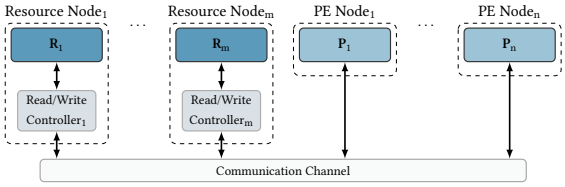
Hardware-supported Systems. *CAP* [58], *StarOS* [36], *IBM System/38* [34], and CHERI [74] extend *instruction set architecture* (ISA) to protect memory in single systems. CAP proposes a hardware design cand an associated operating system to manage capabilities. IBM System/38 implement capabilities 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[70]. CEP [7] 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 [32] 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.

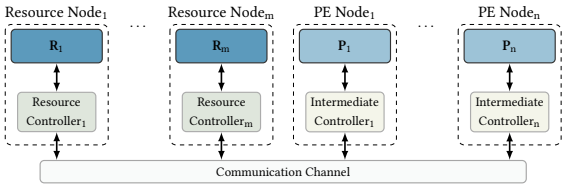
OS-based Systems. *HYDRA* [75] is an OS that treats capabilities as references to resources and allows only the kernel to manipulate them. *KeyKOS* [30], *Accent* [63], *Mach* [1], and *EROS* [67] enforce capabilities via micro-kernels. KeyKOS divides a system into domains and employs capabilities to 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 [1] is an object-oriented OS that enables processes to pass data and capabilities via a messaging system.

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.



(a) Abstract Disaggregated System.



(b) Distributed Access-control System.

Figure 3: Distributed Systems.

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.

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.

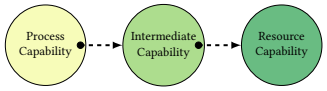


Figure 4: The Capability Model.

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 trees* [59]. A rooted tree is a tree with a particular vertex v as its *root*. We refer to the trees as *capability trees* ($\mathbb{T}_r \cup \mathbb{T}_i$), 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*). Figures 5a to 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 Figure 5a.

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.

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. [50] 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:

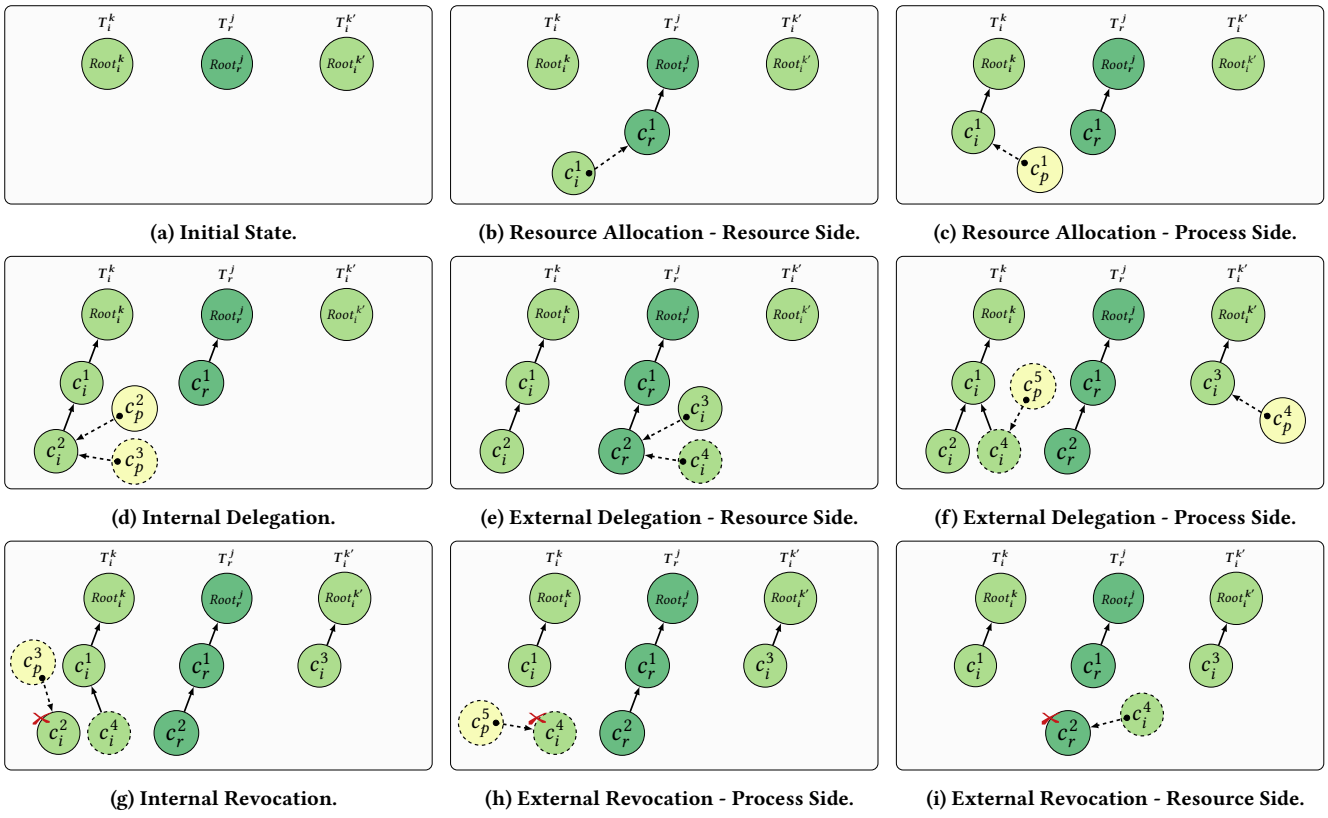
$$c' \prec c \iff rsrc(c') \subseteq rsrc(c) \ \wedge \ priv(c') \subseteq priv(c)$$

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.*



(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

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:

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

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

Due to space constraints, the interested readers can find the proofs of Theorem 5.1 and Theorem 5.2 in Appendix A.1 and Appendix A.2, respectively.

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:

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

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

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

5.2.2 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.

Lemma 1. *(Resource Allocation) Given process $p \in P_k$ and resource $r \in R_j$, p receives a valid process capability, $c_p \in \mathbb{C}_p$, by requesting for allocating resource r such that $rsrc(c_p) = r$ and $priv(c_p) \subseteq priv(c_r^{root_j})$.*

Proof. We model the intermediate and the resource controllers in a resource-allocation operation by programs $I_{ra} \stackrel{\text{def}}{=} (L_{ra}^i,\ T_{ra}^i,\ s_{ra}^i, t_{ra}^i)$ and $R_{ra} \stackrel{\text{def}}{=} (L_{ra}^r, T_{ra}^r, s_{ra}^r, t_{ra}^r)$, 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, and Figures 7a and 7b illustrate sequential diagrams of programs I_{ra} and R_{ra}, respectively. Now, we prove that requesting a resource gives the process a valid process capability using the A-C method (Section 2.2).

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

$$\vdash \langle A_{ra}^i,\ C_{ra}^i \rangle :\ \{\varphi_{ra}^i\}\ I_{ra}\ \{\psi_{ra}^i\}$$

A_{ra}^i, C_{ra}^i, φ_{ra}^i, and ψ_{ra}^i are assumption, commitment, precondition, and postcondition of the program, respectively.

Program I_{ra} has sent or received no message when it enters the start state. After starting, I_{ra} initializes its variables, such as its reference to the capability tree. In addition, the postcondition is set to false, meaning the program iterates over a sequence of its states.

In each iteration, I_{ra} handles a resource-allocation request from the process. The iteration starts when I_{ra} receives the request from the process via channel A. I_{ra} assumes that the *PID* in the request is unique inside the PE node and belongs to the process. The assumption is essential because it guarantees the one-to-one relation between capabilities and processes. I_{ra} creates a resource-allocation request in the next step, inserts the *PID* inside the request, and sends the request to program R_{ra} via channel C. Program I_{ra} commits that the departed requests via channel C contain unique *PID*s.

By receiving the response from R_{ra}, program I_{ra} assumes that it contains a valid intermediate capability. I_{ra} adds the capability to its tree, forges a valid process capability, and sends the process capability to the process via channel B. I_{ra} commits that each leaving response via channel B comprises a valid process capability.

Resource Controller. The A-C formula for program R_{ra} is as follows:

$$\vdash \langle A_{ra}^r,\ C_{ra}^r \rangle :\ \{\varphi_{ra}^r\}\ R_{ra}\ \{\psi_{ra}^r\}$$

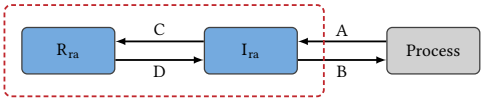


Figure 6: Syntactic Interfaces of the Resource-allocation Programs.

A^r_{ra}, C^r_{ra}, φ^r_{ra}, and ψ^r_{ra} are assumption, commitment, precondition, and postcondition of the program, respectively.

R_{ra} starts while it has sent or received no messages. It receives requests from intermediate controllers via channel C and assumes each request contains a unique *PID* inside the sender's node. After initializing, R_{ra} iterates over a sequence of its states and handles a resource-allocation request in each iteration. By receiving a request, R_{ra} allocates the resource, creates its resource capability, and inserts it inside the capability tree. Furthermore, R_{ra} forges the corresponding intermediate capability and sends it inside a response to the intermediate controller via channel D. Program R_{ra} commits that each response contains a valid intermediate capability.

Parallel Composition. Consider the parallel composition of programs R_{ra} and I_{ra} ($R_{ra}||I_{ra}$). By applying the parallel composition rule (Section 2.2), we deduce the following A-C formula:

$$\vdash \langle A_{ra},\ C_{ra}\rangle :\ \{\varphi_{ra}\}\ R_{ra}\ ||\ I_{ra}\ \{\psi_{ra}\}$$

The parallel-composed program handles all resource-allocation requests from the process. It assumes that each received request via channel A contains a unique *PID* inside the sender's PE node. Furthermore, the parallel composition commits that all the responses it sends via channel B encompass valid process capabilities. □

Lemma 2. *(internal Delegation) Given processes p, $p' \in P_k$, if p delegates its valid capability, c_p, to p', then p' receives a valid capability, c'_p, such that $rsrc(c'_p) \subseteq rsrc(c_p)$ and $priv(c'_p) \subseteq priv(c_p)$*[1].

The second lemma states that the internal delegation of a valid capability by a process creates a valid capability for another process inside the same node.

Lemma 3. *(External Delegation) Given processes $p \in P_k$ and $p' \in P_{k'}$, if p delegates its valid capability, c_p, to p', then p' receives a valid capability, c'_p, such that $rsrc(c'_p) \subseteq rsrc(c_p)$ and $priv(c'_p) \subseteq priv(c_p)$* [1].

The third lemma expresses that externally delegating a valid capability creates a valid capability for the receiver process.

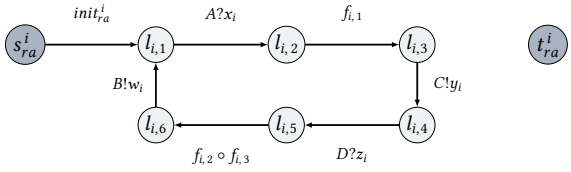
Lemma 4. *(internal Revocation) Given processes p, $p' \in P_k$ and capability $c_p \in \mathbb{C}_p$, delegated from p to p', if p revokes the delegated capability, then c_p becomes invalid* [1].

The fourth lemma indicates that revoking an internally delegated capability by the delegator process will invalidate the delegated capability inside the node.

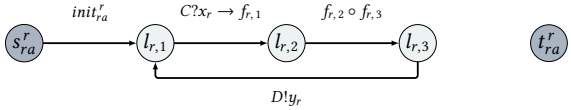
Lemma 5. *(external Revocation) Given processes $p \in P_k$ and $p' \in P_{k'}$, and capability $c_p \in \mathbb{C}_p$, delegated from p to p', if p revokes the delegated capability, then c_p becomes invalid* [1].

The fifth lemma states that revoking an externally delegated capability will invalidate the delegated capability.

[1]The whole proof of Lemmas 1 to 5 can be found in the Technical Report.



(a) Program I_{ra}.



(b) Program R_{ra}.

Figure 7: Resource-allocation Programs. Legend: C?x = the program receives an input from channel C and assigns it to variable x; C!x = the program sends the variable x via channel C.

5.3.1 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:

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

□

5.3.2 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* $p \in \mathbb{P}$:[2]

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

5.4 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. Readers can find the whole proof in the Technical Report, section *Security Properties*.

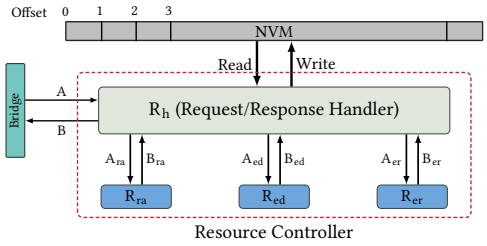
6 USE CASE

MDC [40] 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.

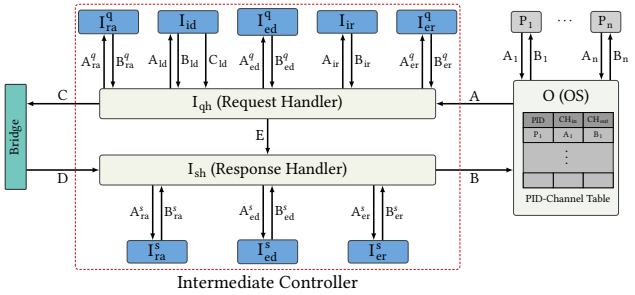
We instantiates our capability-based access-control system for MDC as the use case and demonstrate how it handles requests and capabilities[1]. In our general-purpose system, programs directly connect to other programs via synchronous channels (as depicted in Figure 6). However, in MDC, they are part 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

[1] Appendix B.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 8: Instantiated Access-Control System for MDC.

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 [19]. 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 8a 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 External 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 satisfies 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 8b 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 8b). 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.

7 MODEL CHECKING

We use the SPIN model checker [33] to check our design. SPIN employs the *PROMELA* language to define models. We built the models of Lemmas 1 to 5 using PROMELA, where each model consists of the following PROMELA object types.

Processes: a process is a *proctype* object type that defines the behavior of programs in the lemmas, such as Ira and Rra (Figure 6).

Table 3: Correctness Checks of the Models

Model	Transitions	Depth	Time(sec)	Lines of Code
RA	2224	127	0.01	350
ID	950	106	0.01	349
ED	26554	141	0.06	399
IR	1165	86	0.01	287
ER	26403	140	0.06	380

Channels: a channel models a synchronous messaging channel in the A-C method, using *chan* object type. a channel is a one-way semantic interface between two programs in the lemmas (Figure 6).

Data Objects: PROMELA only supports basic object types, such as *byte* and *short*, but not complex ones, such as *floating-point*.

Claims: a system verification proves what is possible in a model and what is not. PROMELA defines a model's logical correctness using five claim types: *basic assertions*, *meta labels*, *never claims*, and *trace assertions*. We employ basic assertions and trace assertions to check the correctness of our models according to the assumption-commitment method. A Basic assertion includes an expression that PSIN evaluates as False or True during a verification; a False assertion causes the verification to fail. A trace assertion checks if the model has exchanged messages in a specific order and if the exchanged messages include defined data.

We modeled and verified Lemmas 1 to 5, for the general access control system and the MDC instantiated system; SPIN verified all the models as correct. Table 3 depicts the details of each verification—Appendix C contains the modeled programs,I_{ra} and R_{ra}, and the trace assertion for Lemma 1 (resource allocation).

8 CONCLUSION

In this paper, we presented SADRA, a new capability-based access-control system for disaggregated-resource architectures. SADRA provides a fully distributed access-control system on resources and PEs sides and supports fast delegating capabilities and revoking delegation hierarchies using capability trees. We proved that our system guarantees capability and authority safety; thus, our design is sound. Furthermore, we proved that our system supports isolating processes' resources. We designed our system with generality in mind. Thus, other distributed systems can employ it to control access to their shared distributed resources. We demonstrated a use case by instantiating our access-control system for MDC.

Implementation. Applications employ co-processors as processing accelerators [62]. Researchers have designed different systems demonstrating the feasibility of co-processor-based systems using ASICs, FPGAs, and GPUs [14, 24, 27, 37, 41, 64, 68, 69]. Motivated by these solutions, our design presents a practical access-control system for real-world applications.

Limitations and Future Work. We assumed in the use case that an adversary could not comprise the part of an OS that guarantees the process ID in a request matches the process ID of the sender. Furthermore, we assume processes and OSs can work with capabilities. However, both assumptions are strong ones. In future work, we will implement and integrate our system with PE-side capability systems, such as CHERI and L4, and develop a programming model based on our system design.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. (1986).

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[4] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[5] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. *In FAST* (2014).

[6] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 189–203.

[7] Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert NM Watson, Paolo Faraboschi, and Dejan Milojicic. 2019. Memory-side protection with a capability enforcement co-processor. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–26.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 29–44.

[9] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38.

[10] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 1–9.

[11] Kirk M Bresniker, Paolo Faraboschi, Avi Mendelson, Dejan Milojicic, Timothy Roscoe, and Robert NM Watson. 2019. Rack-scale capabilities: fine-grained protection for large-scale memories. *Computer* 52, 2 (2019), 52–62.

[12] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. 2020. Fluidmem: Full, flexible, and fast memory disaggregation for the cloud. In *IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 665–677.

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

[14] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 1–13.

[15] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. 2022. 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*. 434–451.

[16] George W Cox, William M Corwin, Konrad K Lai, and Fred J Pollack. 1981. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proceedings of the eighth ACM symposium on Operating systems principles*. 125–126.

[17] Compute Express Link (CXL). 2023 (accessed April 16, 2023). CXL Specification. https://www.computeexpresslink.org/download-the-specification.

[18] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. 2001. Grid information services for distributed resource sharing. In *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 181–194.

[19] W-P De Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2001. *Concurrency verification: Introduction to compositional and non-compositional methods*. Vol. 54. Cambridge University Press.

[20] Jack B Dennis and Earl C Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

[21] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 103–114.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[23] Robert S. Fabry. 1974. Capability-based addressing. *Commun. ACM* 17, 7 (1974), 403–412.

[24] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[25] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[26] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[27] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 325–336.

[28] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[29] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. 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*. 417–433.

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

[31] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not.. In *ICAC*, Vol. 13. 23–27.

[32] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. Semperos: A distributed capability system. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 709–722.

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

[34] Merle E Houdek, Frank G Soltis, and Roy L Hoffman. 1981. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th annual symposium on Computer Architecture*. 341–348.

[35] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[36] Anita K Jones, Robert J Chansler Jr, Ivor Durham, Karsten Schwans, and Steven R Vegdahl. 1979. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the seventh ACM symposium on Operating systems principles*. 117–127.

[37] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[38] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, Kobi Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 690–695.

[39] Kimberly Keeton. 2015. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, Vol. 10.

[40] Kimberly Keeton. 2017. Memory-Driven Computing.. In *FAST*.

[41] Kurt Keutzer, Sharad Malik, and A Richard Newton. 2002. From ASIC to ASIP: The next design discontinuity. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 84–90.

[42] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2021. Deact: Architecture-aware virtual memory support for fabric attached memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 453–466.

[43] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 488–504.

[44] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, et al. 2017. Understanding {Rack-Scale} Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[45] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. 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*. 574–587.
[46] Jochen Liedtke. 1995. On micro-kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.
[47] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.
[48] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.
[49] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated data centers: Challenges and trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26.
[50] Sergio Maffeis, John C Mitchell, and Ankur Taly. 2010. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 125–140.
[51] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*.
[52] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. 2003. *Capability myths demolished*. Technical Report. Technical Report SRL2003-02, Johns Hopkins University Systems Research ....
[53] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 106–122.
[54] Dave Minturn. 2015. Nvm express over fabrics. In *11th Annual OpenFabrics International OFS Developers' Workshop*.
[55] Jayadev Misra and K. Mani Chandy. 1981. Proofs of networks of processes. *IEEE transactions on software engineering* 4 (1981), 417–426.
[56] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53.
[57] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage.. In *NSDI, Volume 17*. 17–33.
[58] Roger M Needham and Robin DH Walker. 1977. The Cambridge CAP computer and its protection system. *ACM SIGOPS Operating Systems Review* 11, 5 (1977), 1–10.
[59] Richard Otter. 1948. The number of trees. *Annals of Mathematics* (1948), 583–599.
[60] Antonios D Papaioannou, Reza Nejabati, and Dimitra Simeonidou. 2016. The benefits of a disaggregated data centre: A resource allocation approach. In *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–7.
[61] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. 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)*. IEEE, 868–880.
[62] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.
[63] Richard F Rashid and George G Robertson. 1981. Accent: A communication oriented network operating system kernel. *ACM SIGOPS Operating Systems Review* 15, 5 (1981), 64–75.
[64] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 233–248.
[65] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. 1992. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*. Seattle WA (USA), 39–70.
[66] Jerome H Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.
[67] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*. 170–185.
[68] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks.. In *NSDI*. 255–270.
[69] Hayden Kwok-Hay So and Robert Brodersen. 2008. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 2 (2008), 1–28.
[70] Richard J Swan, Samuel H Fuller, and Daniel P Siewiorek. 1977. Cm* a modular, multi-microprocessor. In *Proceedings of the June 13-16, 1977, national computer conference*. 637–644.
[71] Shin-Yeh Tsai and Yiying Zhang. 2019. 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*. 3–3.
[72] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. *arXiv preprint arXiv:2211.02682* (2022).
[73] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 261–280.
[74] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.
[75] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. 1974. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM* 17, 6 (1974), 337–345.

A PROOFS

A.1 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.

□

A.2 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.

□

A.3 Lemma 1 (Resource Allocation)

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. We prove that by requesting a resource, the process receives a valid process capability.

A.3.1 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 intermediate capability from the resource controller by inserting the capability inside the tree. In addition, it creates the corresponding process capability for the inserted intermediate capability. Finally, it creates a response for the process and sends the process capability 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

A.3.2 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 $\varphi_r, \psi_r, A^r_{ra}$, and C^r_{ra} are as follows:

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

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

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

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

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

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

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

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

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

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

We check the following implications to prove that the above assertion network is an A-C-inductive assertion network w.r.t. assumption A^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.
- $\models Q_{s^r_{ra}} \wedge A^r_{ra} \rightarrow Q_{l_{r,1}} \circ init^r_{ra}$. 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 $isRaReq(last(C))$ and $isUniquePID(last(C).spid)$ predicates. Function $f_{r,1}$ allocates the requested resource and assigns the returned valid r-cap to $rCap$. Hence, Q_{r_2} satisfies $isValidCap(rCap)$ 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 $\sigma \models isValidCap(rCap)$, we have $\sigma' \models isValidCap(iCap)$. Function $f_{i,3}$ creates a resource-allocation response for program I_{ra} and assigns it to y_r, such that $isRaRes(y_r) = true$ and $y_r.cap = iCap$. In addition, because $isValidCap(iCap)$, we have $isValidCap(y_r.cap)$. Thus, this verification holds.
- $\models Q_{l_{r,3}} \wedge A^r_{ra} \rightarrow ((A^r_{ra} \rightarrow Q_{l_{r,1}}) \wedge C^r_{ra}) \circ g$, 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.

A.3.3 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 $R_{ra} \,||\, I_{ra}$. By applying the parallel composition rule, we deduce the following A-C formula:

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

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

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

$$C_{ra} \stackrel{\text{def}}{=} \#A = \#B = \#C = \#D > 0 \rightarrow (isRaRes(last(B)) \wedge isValidCap(last(B).cap))$$

type (8 bits)	**permissions** (16 bits)		**capability length** (32 bits)
process ID (32 bits)			**PE node ID** (32 bits)
size (64 bits)			
list of (base, length) pairs (64 bits, 64 bits)			

(a) Resource Capability Structure.

type (8 bits)	**permissions** (16 bits)	**indicator** (8 bits)	**process ID** (32 bits)
resource node ID (32 bits)			**capability pointer** (32 bits)
base (64 bits)			
length (64 bits)			
hash value (62 bits)			

(b) Intermediate and Process Capabilities Structure.

Figure 9: Capability Structure.

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, $R_{ra} \,||\, I_{ra}$ generates a valid capability when it receives a resource-allocation request from a process.

□

B USE CASE

B.1 Capability Structures

In this section, we describe the structure of each capability type for MDC and how they handle addressing memories as resources.

Resource Capability. Figure 9a illustrates resource capability structure. The first field specifies the capability's type. The *permissions* field indicates the process's access rights, in which each bit maps to a specific permission. The *process ID* (*PID*) and *PE node ID* (*PNID*) fields bind the capability to a particular process. The size field indicates the size of the allocated memory.

MDC is a disaggregated-memory architecture. The memory manager may allocate separate sections to respond to a memory-allocation request. Therefore, the resource controller must address all the sections within a resource capability. Each pair define the start and the size of a memory section. After the controller stores the list in the capability, it calculates the capability size and stores the result in the capability length field.

Intermediate and Process Capability. Figure 9b depicts intermediate and process capability structure. The type, permissions, and *PID* fields are the same as in the resource capability. The capability pointer points to the corresponding resource or intermediate capability inside a capability tree. The intermediate controller locates the resource node using the *resource node ID* (*RNID*). In addition, if the controller decides to mask the resource node from the process, it sets the field to zero in the process capability when forging it.

C MODEL CHECKING CODES

```
active proctype Intermediate_Controller()
{
    initICTree();

    int initialValidCaps = 0;
    int finalValidCaps = 0;
    short iCapPtr;
    mtype:msgStatus msgStts;
    ProcessRequestDetails pReqDtails;
    IntCtrlRequestDetails iReqDtails;
    IntermediateCap intCap;
    ProcessCap pCap;

    validIntCaps(initialValidCaps);
    assert(initialValidCaps == 1);

    A?prcRaReq(msgStts, pReqDtails);
    assert(pReqDtails.procId == PROCESSID);

    msgStts = dontcare;
    genIntRaReq(pReqDtails,iReqDtails);
    C!intRaReq(msgStts, iReqDtails);

    D?resRaRes(msgStts, intCap);
    assert(msgStts == success);
    assert(intCap.procId == PROCESSID);

    insertIntCap(intCap, iCapPtr);

    validIntCaps(finalValidCaps);
    assert(finalValidCaps == initialValidCaps + 1);

    genProccessCap(iCapPtr, pCap);
    assert(pCap.procId == PROCESSID)

    B!intRaRes(success, pCap);
}
```

Resource Allocation–Intermediate Controller.pml

```
active proctype Resource_Controller()
{
    initRCTree();

    int initialValidCaps = 0;
    int finalValidCaps = 0;
    short rCapPtr;
    mtype:msgStatus msgStts;
    IntCtrlRequestDetails iReqDtails;
    ResourceCap resCap;
    IntermediateCap iCap;

    validResCaps(initialValidCaps);
    assert(initialValidCaps == 1);

    C?intRaReq(msgStts, iReqDtails);
    assert(iReqDtails.procId == PROCESSID);

    createResCaps(iReqDtails, rCapPtr);
    validResCaps(finalValidCaps);
    assert(finalValidCaps == initialValidCaps + 1);

    genIntermediateCaps(rCapPtr, iCap);
    assert(iCap.procId == PROCESSID);
    assert(iCap.resCapPtr == rCapPtr);

    D!resRaRes(success, iCap);

}
```

Resource Allocation–Resource Controller.pml

```
trace {
S0:
   if
    :: A!prcRaReq,_ -> goto S1;
   fi;
S1:
   if
    :: A?prcRaReq,_ -> goto S2;
   fi;
S2:
   if
    :: C!intRaReq,_ -> goto S3;
   fi;
S3:
   if
    :: C?intRaReq,_ -> goto S4;
   fi;
S4:
   if
    :: D!resRaRes,success,_ -> goto S5;
   fi;
S5:
   if
    :: D?resRaRes,success,_ -> goto S6;
   fi;
S6:
   if
    :: B!intRaRes,success,_ -> goto S7;
   fi;
S7:
   if
    :: B?intRaRes,success,_ ;
   fi;
}
```

Resource Allocation–Trace Assertion.pml