

Enabling Cloud-Scale Distributed Capabilities

Otto White
Imperial College London
London, United Kingdom
otto.white20@imperial.ac.uk

Yaoxin Jing
Imperial College London
London, United Kingdom
y.jing24@imperial.ac.uk

Adrien Ghosn
Microsoft Azure Research
Cambridge, United Kingdom
ghosn.adrien@gmail.com

Michael Steiner
Intel Labs
Hillsboro, USA
michael.steiner@intel.com

Anjo Vahldiek-Oberwagner
Intel Labs
Berlin, Germany
anjovahldiek@gmail.com

Mona Vij
Intel Labs
Hillsboro, USA
mona.vij@intel.com

Lluis Vilanova
Imperial College London
London, United Kingdom
vilanova@imperial.ac.uk

Abstract

Modern applications rely on service-oriented architectures to increase development productivity, cost-effectiveness, and scalability. However, the growing complexity of cloud stacks, driven by multi-tenancy, multi-party computations, and dynamic service collaboration, introduces security risks stemming from over-privileged access. While enforcing the principle of least authority (PoLA) mitigates these risks, implementing PoLA at scale is prohibitively complex and costly. If we instead look at existing access control systems, such as RBAC [25] or ABAC [36] at the application layer or security groups [14] at the network layer, they rely on externally defined policies, provide limited abstractions, and require retrofitting security onto applications, leading to over-privilege.

Conversely, capability-based security offers an application-driven solution for access control, leading to tight integration of security with application semantics, and making PoLA attainable. We analyse existing capability systems and find that they fall short at cloud-scale due to limitations in performance, scalability, or fault tolerance. We present a distributed capability system that through a sharded, decentralised architecture, capability versioning, and application-defined revocability, enables microsecond-scale delegation and revocation, data center scale scalability, and fault-tolerance. Our evaluation demonstrates capability operation latency and system-wide resource consumption scale better than previous capability systems, at μsecond-scale latency.

CCS Concepts

• **Security and privacy** → **Access control**; • **Computer systems organization** → **Cloud computing**; Reliability; • **General and reference** → Performance.

Keywords

Access Control Systems, Distributed Capabilities, Cloud Computing, Performance, Scalability, Fault-Tolerance, Reliability



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

HCDS '25, Rotterdam, Netherlands

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1470-2/25/03
https://doi.org/10.1145/3723851.3723854

ACM Reference Format:
Otto White, Yaoxin Jing, Adrien Ghosn, Michael Steiner, Anjo Vahldiek-Oberwagner, Mona Vij, and Lluis Vilanova. 2025. Enabling Cloud-Scale Distributed Capabilities. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25), March 30, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3723851.3723854

1 Introduction

Cloud-native applications use service-oriented architectures for elasticity, productivity, and cost-effectiveness. Standalone services — such as databases [7, 8, 28], data caching [20], and ML-as-a-service [1–3] — are composed [10] to rapidly build complex processing pipelines and systems with advanced functionalities. For example, an image-based ML inference application can be rapidly built by combining a request ingestion service (e.g., Nginx), an inference service using vLLM, and a storage service like Amazon S3 for user input files.

Despite its advantages, service-oriented architectures often lag in security. Their modularity, flexibility, and composability often lead to "*over-privilege*". Services are granted excessive access without strict controls on resources or network interactions. This over-privilege increases security risks, including compromised credentials, insider threats, accidental misuse [23, 24], and cross-party exposure in multi-tenant environments [15]. For example, in the ML inference scenario above, the ingestion service may only allow requests referencing authorized input files. However, an attacker could craft a malformed inference input to bypass this check, exploiting an over-privileged inference service, which has access to the entire storage, to retrieve unauthorized data.

From a security perspective, applications should follow the principle of least authority (PoLA) [22], granting services only the minimum privileges needed to complete their current task. However, achieving PoLA at cloud scale is currently impractical because: (i) security is retrofitted onto applications with externally defined policies, rather than being a first-class citizen of the application development process; and (ii) it is costly to tightly secure and maintain realistic applications using policies developed independently from the applications [9, 35]. An application-driven PoLA approach would provide a more effective path to robust security.

Capability-based security is ideal for enforcing application-driven PoLA, but no existing capability system can operate at cloud scale.

The security benefits of capabilities are well established, including privilege minimization, mitigation of confused deputy attacks, and elimination of ambient authority [18, 21]. Each capability is an unforgeable token that allows its holder to securely "invoke" a specific operation. In turn, the holder of a capability can securely and dynamically "delegate" it to another party, who then holds a copy. The original holder remains, however, in control and can later "revoke" the delegation, ensuring that the delegated permissions are no longer valid. In the ML inference example, application-driven PoLA can be implemented by replacing RPCs with capability invocations [29]. The ingestion service would convert authorized file references into capabilities, delegating them to the inference service and revoking them after the computation. This approach enables flexible computations while ensuring security invariants across services and minimizing per-task privileges.

Unfortunately, previous capability systems cannot reach cloud scale because they cannot address the following challenges: (i) *critical-path performance*, by introducing excessive capability-related network messages (e.g., during RPC invocation), or slow network reconfiguration; (ii) *scalability*, by having capability book-keeping overheads that increase quadratically with the system size; or (iii) *fault tolerance*, by treating the entire capability system as a single failure domain. Previous capability systems exhibit one or more of these problems, either because they were targeting a different scale or as a trade-off between these three challenges.

In this paper, we present a distributed capability system that, to the best of our knowledge, is the first to simultaneously address all three challenges: critical-path performance, scalability, and fault tolerance. As a result, we achieve constant-complexity capability operations at the μsecond scale, regardless of system size, including immediate revocation and with minimal, constant-cost management overheads. We present the design of this new system, which provides the following technical contributions (see § 3):

(1) Self-contained guarded capability sets provide critical-path performance by minimizing costs associated with delegation and revocation operations. When looking at application semantics, only a subset of the delegated capabilities is directly revoked. We therefore introduce the concept of *guarded capability sets*, wherein capabilities can be delegated without updating any central structure. Within a guarded set, the contents of a capability are directly serialized as part of an application RPC (i.e., capability invocation), and a revocation is immediately applied to all capabilities within the same set. We only need to track the creation of such "capability guards", which is a much less frequent operation.

(2) Owner-based metadata sharding provides fault tolerance, ensuring that the failure of one node in the system does not affect other nodes. We observe that capability authorization can be safely delayed until invocation, and that invocation should always be possible as long as the invoked application or the node where it runs have not failed. We therefore co-locate metadata needed for invocation authorization (namely capability guards), with the application that the capability points to (the capability "owner"), ensuring that failure of any other application or node will not affect failure tolerance. This sharding also helps naturally distribute capability operations across nodes, eliminating bottlenecks and facilitating scalability as well as fault tolerance.

(3) Metadata versioning provides the ability to reach cloud scale with constant system overheads on each node. Capabilities can be seen as indices to a global, distributed space, wherein delegation and revocation provide a second, temporal dimension. We directly represent this temporal dimension by storing a version number for the node and guard each capability points to. As a result, we avoid the costly broadcast operations that other systems need to "clean up" capabilities when they are revoked, or when an application or node fail. The versions are large enough that version cleanup can be performed in the background at negligible cost, even at very large system sizes.

We provide a careful analysis of previous capability systems and how they fail to deliver on the various challenges we identified above (see § 2). We also demonstrate how our system delivers on all three challenges by evaluating a prototype using a small-scale data center, as well as modelling it on larger-scale systems (see sections 3 to 5).

2 Motivation

Capabilities have been extensively studied as an effective mechanism to enforce PoLA and enable fine-grained security [18]. A principled shift to a capability-based cloud is now timely, as the increasing complexity of cloud applications is pushing existing security approaches to their limits (§ 2.1).

Nevertheless, capability systems have seen little mainstream adoption. They are not as well understood as other security mechanisms we use today, even though interest in them is growing (§ 2.2). In addition, adopting capabilities requires careful consideration of prior designs, leveraging their insights while addressing their unforeseen limitations when deployed at cloud scale (§ 2.3).

2.1 Limitations of non-capability systems

Traditional security enforcement mechanisms, such as RBAC and ABAC, fail to achieve PoLA under the scale and complexity of modern cloud systems. They define policies that are *decoupled from application semantics*, leading to independent policy creation, maintenance, and synchronisation. Their opinionated abstractions (roles and attributes) either struggle to capture intricate semantics, dynamicity, and collaboration, or their policies become excessively complex when doing so (e.g., role-explosion for RBAC [9] or massive privilege spaces for ABAC [24]). Policy mining attempts to automate the creation of complex, least-privilege security policies but only tackles a symptom of decoupled security abstractions.

2.2 A primer on capability systems

Capability systems allow the secure *creation* of capabilities, preventing forgery and tampering. This is typically achieved by either using: (i) cryptographic integrity checks [19]; (ii) a partitioned state maintained by the OS kernel [5, 12] or (iii) a ISA-controlled in-memory representation [32].

Capability systems also provide the following operations: (i) *invocation* performs an operation based on the rights encoded in the capability — e.g., send a message to another process, or access a specific memory address; (ii) *delegation* allows a process to delegate the authority to invoke that capability to another process by sending it a copy of a capability it holds — this describes a sender/receiver

System	Intended scale	Critical-path performance	Scalability	Fault tolerance
Barrelfish [11]	Distributed OS	✗	✗	✗
SemperOS [13]		✗	✓	✗
Amoeba [19]		✓(*)	✓	✗
CapNet [6]	Data center	✗	✗	✗
FractOS [29]		✓	✗	✗
BlendCAC [34]	World-wide IoT	✗	✓	✓
This work	Data center	✓	✓	✓

Table 1: Comparison of various capability systems across the three challenges for cloud-scale distributed capabilities. (*) Design limitations only partially fulfill a challenge.

relationship that is tracked as the *capability derivation tree (CDT)*; (iii) *diminishing* permissions allows delegating a capability to oneself with monotonically decreasing permissions [30]; and (iv) *revocation* allows the holder of a capability to invalidate all capabilities within the CDT subtree rooted at the capability being revoked. One can combine delegation with permission diminishing to, for example, delegate read-only permissions for a file capability that is held with read-write permissions.

Revoking a CDT subtree is known as *selective revocation* and is key for applications to precisely control delegated authority, but it is also arguably the most complex operation on any capability system. Note that some form of revocation is always necessary: reusing resources authorized through a CDT (e.g., a memory allocation) before the CDT has been fully revoked will lead to use-after-free security problems. Due to this complexity, some systems instead provide non-selective revocation [17] or avoid reuse through a trusted, global garbage collection pass for capabilities [31, 33].

2.3 Challenges of cloud-scale capabilities

We cannot easily shift previous capability systems into the cloud because the larger scale, greater networking costs, and higher failure rates put unanticipated pressure on their design. No existing design fully resolves the following three challenges simultaneously, favoring one aspect over another: (i) *critical-path performance* describes whether the capability operations in § 2.2 are fast enough to be used on the hot path of applications; (ii) *scalability* ensures that the latency and management overheads of all operations are constant for each node as we increase the system size (i.e., linear with system size) — failure to do so leads to *centralization and broadcast bottlenecks*, introducing unreasonable inefficiencies for both applications and operators; and (iii) *fault tolerance* is critical at cloud scale, given that capabilities are linked to both security and system utilization, and that cloud-scale systems are known to have large aggregate failure rates [4].

The CDT is one of the most important parts of a capability system with selective revocation, as it is updated during creation, delegation, and revocation and checked during invocation. As a result, the specific implementation of a CDT has a fundamental impact on all three challenges. In the rest of this section, we analyze how relevant capability systems measure up to these challenges, as summarized in Tab. 1.

Critical-path performance. Many previous systems cannot perform at larger scales; e.g., Barrelfish [5] and SemperOS [13] were designed to execute within a chip. Networked operations become orders of magnitude slower, often leading application developers to avoid them by *over-permissioning* applications and, therefore, failing to follow PoLA.

Interestingly, systems will use a distributed CDT to avoid delegation overheads, but distributed CDT traversal leads to slower revocations and, therefore, to *system underutilization* as resources cannot be reused until a revocation is globally consistent (e.g., Barrelfish and SemperOS). For a subtree of size C, selective revocation has complexity $O(C)$, leading to up to as many network messages for CDT traversal.

Other systems rely on technologies that are too slow for critical-path performance. CapNet [6] enforces delegation and revocation by updating SDN flow-tables, which can take 10–100 msec [16]. Similarly, BlendCAC [34] relies on blockchain technology with large and unpredictable latency.

In contrast, FractOS [29] and Amoeba [19, 27] can offer critical-path performance by embedding capability operations within application RPCs. FractOS uses partitioned capabilities and supports all capability operations with at most one network message. Amoeba uses cryptographic capabilities; with a poorer permissioning system (e.g., cannot support diminishing memory ranges), some operations can require multiple cryptographic passes, and can only prevent capability forgery at ingress (leading to DoS attacks).

Scalability. With tens or hundreds of thousands of nodes and an equally large number of applications, in a cloud-scale capability system with N nodes and K capability ops/sec/node, a centralized solution is a bottleneck: a single node must process $O(NK)$ operations. This is the case for CapNet, which does central updates during delegation and revocation.

Distributed solutions such as Barrelfish and FractOS have a broadcast bottleneck during revocation. Barrelfish uses a two-phase commit to all nodes, which has quadratic overheads with $O(N^2K)$ messages, and lowers system utilization by delaying resource release. In contrast, FractOS uses an indirection table to immediately release resources but needs a broadcast operation to (eventually) reuse table entries, again with quadratic overheads of $O(N^2K)$.

Some systems exist that can reach cloud-scale. BlendCAC leverages blockchain, which requires excessive computation with long latencies, while the number of miners can further degrade its efficiency. Amoeba employs cryptographic capabilities checked by the serving application when invoked, which naturally avoids centralization and broadcast bottlenecks. SemperOS distributes its CDT across multiple micro-kernel instances and avoids broadcast bottlenecks for selective revocation via distributed traversal.

Fault tolerance. A centralized design such as CapNet puts the entire system in a single failure domain, whereas a distributed CDT such as in Barrelfish and SemperOS puts multiple nodes in a single failure domain. This is because failure on any node along the CDT will partition the data structure and strand its resources. FractOS uses CDT sharding to avoid both problems but cannot handle node failures reliably, whereas Amoeba has similar reliability problems.

Replication is not adequate either, as it reduces critical-path performance; e.g., BlendCAC leverages blockchain, which is orders of magnitude slower than an application RPC.

3 Design

We now describe the design of our distributed capability system, which simultaneously addresses all three cloud-scale challenges (see § 2.3). Our system is an extension of FractOS [29], a distributed multi-kernel capability system that targets heterogeneous, disaggregated data centers. It is only able to provide critical path performance, and we modify it to address all three challenges.

3.1 Application-visible interface

As depicted in Fig. 1, the kernel is deployed as per-node kernel instances known as *controllers*. Each *process* can only interact with the capabilities it holds via its assigned controller, which implements partitioned capabilities; i.e., capabilities are indices to a per-process *capability table* (see userspace application in Fig. 1, and § 2.2).

A capability invocation is routed through the controller network and into the "owner" process that created the root of that capability's CDT. For example, the `Application` in Fig. 1 invokes `Cap7` to send a message to `SSD Service`, who is the owner of that capability as well as the `SSD1` object it refers to, which is routed through controllers 2 and 3 (the invoking and the owning controllers, respectively).

Our system provides the same capability types as FractOS: *memory* and *request*. A *memory* capability represents a memory region within the owning process and provides remote access to it. It encodes an access mode and address range, which can be diminished (e.g., go from read/write to read-only, or authorize a smaller memory buffer). A *request* capability represents the ability to invoke some remote functionality, such as an access to `SSD1` for `Cap7` in Fig. 1. Request invocations can be parameterised with other capabilities that are delegated to the owning process, as well as with a series of data buffers whose contents are sent as part of an invocation. These request "arguments" are always received by an invocation handler function in the capability's owning process; they can be used not only to delegate access to other services but also to specify a "continuation" request to be executed upon completion of the invocation handler. This can be used by applications to implement any communication model, such as unidirectional RPCs, synchronous RPCs, or even distributed dataflow.

3.2 Capability system key design ideas

Self-contained guarded capability sets. Capabilities can be derived, invoked, and delegated without contacting additional processes or controllers. The internal representation of a capability (e.g., the arguments in a request) is entirely self-contained within the controller assigned to the process holding that capability. Every delegation creates a new copy of that representation (with modified contents when diminishing it or when adding request arguments).

To avoid the cost and complexity of selective revocation, we provide it only when applications need it, as exemplified in Fig. 2. Instead of representing the full CDT, processes can explicitly identify which capabilities need to be selectively revocable (the root of each subtree on the left of Fig. 2). We call these potential revocation points *capability guards* (or guards, for short), and replace the traditional CDT with a guard derivation tree (GDT), as shown on the right of Fig. 2. The GDT is never updated when capabilities are derived or delegated (unlike a CDT); instead, the GDT is only updated when a new *revocable* capability (i.e., a guard) is explicitly requested by an application (e.g., `c1` and `c2` are revocable capabilities with respective guards `g1` and `g2`). This ensures the GDT is embedded within the conceptual CDT, while being a lot smaller as it describes only the guards needed by applications, instead of all delegations.

During the invocation of `Cap7` (top-center of Fig. 1), `Controller2Ver1` holds a reference to the owner's GDT (owning controller/guard tuple `(Controller3Ver2, Guard2Ver3)`), which is used to send a message to the owning `Controller3Ver2`, who in turn uses its GDT to send the invocation information to the owning process via `Guard2Ver3`. Revoking the root capability of a guarded subtree (e.g., `c2` in Fig. 2) effectively revokes all derived capabilities (`c4`, `c5`) that point to the same guard (`g2`). Note that when a guard is revoked, the owning controller will reject invocations for it and recursively revoke its child guards, effectively providing the revocation semantics of a CDT.

Owner-based metadata sharding. A GDT is stored within the owning controller, which is co-located with the owning process. Any controller holding a capability (on behalf of one of its processes) that is owned by another controller simply holds a guard reference (the bottom-left `Cap` circle in Fig. 1).

This naturally shards physical resources used by processes, together with capability metadata. GDTs are distributed across the system, but all invocations can always be checked for revocation against the owning controller, which must always be contacted for an invocation (e.g., to deliver a request invocation).

Metadata versioning. All guards owned by a controller are stored in its *guard table*, a finite-size data structure in memory. A guard is globally referenced by other controllers using a tuple that contains a *controller identifier* (using the owning node's address) and a *guard identifier* (using an index to the owner's guard table).

Without delegation tracking or revocation broadcasts (which fail to address the challenges in § 2.3), naively reusing a guard table entry, or rebooting a node, will result to the same use-after-free problems we face with traditional CDTs (see § 2.2).

We therefore introduce guard and controller versioning. Guard identifiers have a 64-bit unsigned version that is incremented before a guard table entry is reused, and controller identifier versions are similarly incremented each time a node is (re)booted (e.g., after a node failure). Invocation of a revoked capability (or after a node reboot) will therefore be rejected as the versions will not match.

The system can rarely run out of version numbers. Guard versions increments are independent between nodes, and a node revoking capabilities at line rate will run out guard versions after 4×10^{12} years (10^9 revocations/sec, with a guard table of 8×2^{10} entries and 64-bit versions). Similarly, a node with a typical 0.1% failure rate will run out of controller versions after 2×10^{18} years (64-bit versions). We split the guard version space of a controller into ranges, select live guards in the oldest version range, assign them new versions in the youngest available range, broadcast this change to all controllers as a version upgrade request, and finally mark the range as free (the same approach applies for node versions). This results in negligible CPU and network overheads since guards are often revoked before an upgrade is needed, and we can pace upgrade messages across a vast timeline.

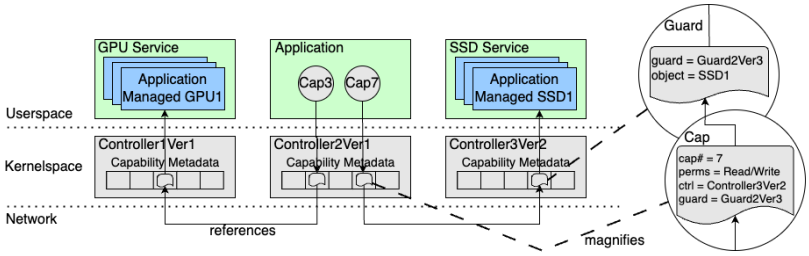


Figure 1: FractOS distributed capabilities architecture

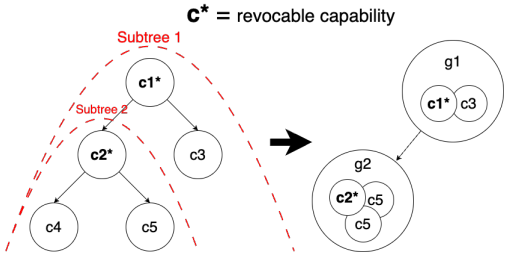


Figure 2: Relationship between CDT and GDT.

3.3 Discussion on cloud-scale challenges

Given the three key design ideas of our capability system, we can now discuss how they fully address the three challenges for building a cloud-scale distributed capability system.

Critical-path performance. Our system has constant CPU complexity and no additional network messages for invocation and delegation, since capabilities are self-contained with their remote guard references (there are no CDT updates), and piggy-back on application RPCs. Capability creation is always local and has a constant cost. Furthermore, the use of guards allows processes to locally "close" their capabilities without performing any revocation (i.e., remove a capability from the process' capability table). When a single guard is revoked, a network roundtrip message to the owning node is required, which simply invalidates a guard table entry.

Scalability. Our system eliminates both centralisation and broadcast bottlenecks, unlike most of the previous systems. Using guards ensures that delegation never updates the corresponding GDT. Owner-based GDT sharding brings various benefits: (i) localizes GDT management on each owning controller (as opposed to other systems with distributed CDT management); (ii) provides a simple serialization point for managing and checking authorization; and (iii) gracefully distributes load between controllers according to the load served by their respective processes. Finally, guard versioning ensures that immediate and selective revocation can be done with a single network roundtrip to the guard's owning controller (unlike previous work, which uses broadcast operations or system-wide garbage collection).

Fault tolerance. Our system maintains per-node failure domains and handles failure in a secure way. Owner-based metadata sharding stores all capability metadata, and handles all capability operations for a resource on the owning controller. This reduces the failure domain for the resource to a single node. Existing systems either expand failure domains to all nodes that participate in a distributed CDT [13], or treat the entire system as a failure domain [5]. Although node failure is inevitable and can still affect service availability, controller versioning in guard references ensures that applications perceive node failure equivalently to if the failed controller revoked all of its capabilities. This enables applications to safely handle the expected behaviour that their capability was revoked, and trigger error handling code such as re-acquiring a new capability from an available service. Nevertheless, this does not degrade the system's fault tolerance, as handling such errors is up to the applications, similar to how they must already handle network connection errors today.

4 Implementation

To build our system, we extended FractOS [29] to address all three cloud-scale challenges. Processes can only issue controller syscalls via asynchronous local network messages; both are deployed as Linux processes, and controller instances are cataloged by a cluster *manager*. We modified the controllers and manager to introduce 64-bit versions to both controller identifiers and guard table entries/references (making capability references globally unique and versioned). To decrease the frequency of guard version increments, we batch guard table entry releases and monotonically increase a guard version counter with each batch. We also modified the controllers to move all control plane operations to a separate thread (e.g., handling guard batches or remote node failures), which improves application tail latency.

5 Evaluation

We evaluate the critical-path performance of our system, and study its scalability using both a small prototype cluster and a mathematical model for larger cluster sizes. We do not quantify fault tolerance; as discussed in § 3 and § 4, the design provides per-node failure domains, and enables applications to handle service failures as expected behaviour (equivalent to revocations).

5.1 Methodology

We perform all measurements in a 7-node cluster where nodes have 2× Intel Xeon E5-2630 v4 CPUs, 64–92GB DRAM and 1× 100 Gbps Mellanox ConnectX-4. Our larger 63-node cluster is emulated by deploying up to 63 FractOS controllers across all physical nodes, and pinning all system and application threads to separate CPUs and NUMA-aware allocations.

5.2 Critical-path performance: Capability operations

We have measured the latency of core capability operations on our system to determine if they are suitable for the critical path of applications. The results are summarized as a breakdown of the main operations in Fig. 3, with experiments repeated until having a standard deviation below 3% of the mean, with 2σ confidence. They include creation of a single capability (*Create*), invocation of a two-way RPC that goes across two controllers (*Empty RPC*), and two more RPC variants where a single capability is delegated (*RPC w/ delegate*) and where a capability is created, delegated and finally revoked at the end of the RPC (*RPC w/ delegate+revoke*).

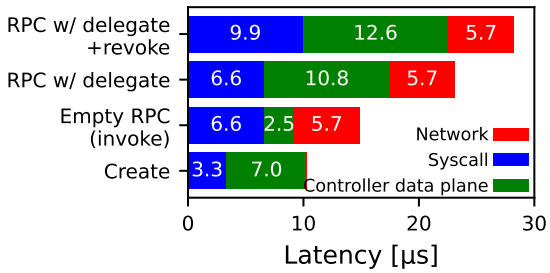


Figure 3: Latency of core capability operations.

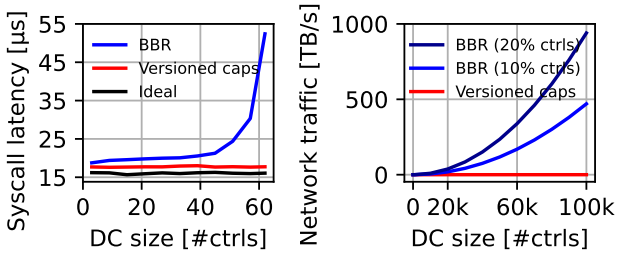


Figure 4: Application latency and network overheads for the prototype cluster (left) and large-scale model (right).

The last variant is particularly important as it allows us to perform a secure RPC that can only be responded to once. This approach strictly follows PoLA and has performance independent of system size (see below), while providing well within existing (unsecured) RPC solutions [26]. These results hold at the 99th percentile with 10 K ops/sec on each node for the more complex RPC, and with 200 K ops/sec for the empty RPC (not shown for brevity).

Conclusions: All operations have performance within existing application-level RPC solutions. Unlike other capability systems with a distributed CDT, we can efficiently piggyback all capability operations on existing application RPCs without impacting their performance.

5.3 Scalability: Capability revocation

We have measured the performance of our versioned capability implementation, and compared it against an implementation on the same system using broadcast-based revocation (BBR; present in Barrelfish and FractOS); we do not compare against other designs with centralized or distributed CDTs, as they impact fault tolerance (see § 2).

Latency measurements: The results for both designs are shown in Fig. 4 (left), together with an ideal design where revocations have no performance impact. Revocation is the most important operation for scalability on these designs, and we measure the interference it introduces on a simple application that executed empty syscalls at 10 K ops/sec; on the same node, we place an application that continuously delegates and revokes a capability to another application in a remote node, also at 10 K ops/sec.

We can observe that the revocation overheads of BBR increase with system size and cannot reach data center scale, since the controller is increasingly busy doing broadcasts. In comparison, versioned capabilities have a constant cost regardless of system size, only slightly above the ideal case (due to sub-optimal cache coherency traffic in the controller). A similar measure on all other controllers shows that BBR has a constant application-visible overhead for processing the incoming broadcasts, whereas we have negligible overhead.

Network Transmission: We have also measured the network traffic introduced by revocations using our virtual cluster and used that as a base to model a cloud-scale system. The results are shown in Fig. 4 (right), using the same application configuration and revocation protocols, and show the aggregate network traffic used by the revocation operations as we vary the number of nodes doing revocations concurrently.

We can see that traffic with BBR increases quadratically with the system size, and reaches close to 1,000 TB/sec with only 20% of the nodes doing revocations. In comparison, versioned capabilities introduce no network traffic whatsoever.

CPU Overheads: Similarly, CPU time for BBR increases quadratically with system size, whereas for versioned capabilities, it grows linearly with the number of revocations.

Conclusions: Capability versioning provides support for selective revocation at constant, negligible cost per node, regardless of system size. This is unlike other approaches such as BBR, while at the same time, versioning provides strong fault tolerance properties.

6 Conclusions

We discussed the necessity of adopting cloud-scale PoLA to build secure cloud applications, how capabilities are key to effectively achieve that, and tackled the challenges that such a distributed capability system faces. We analyzed existing systems, identifying their shortcomings in critical-path performance, scalability, and fault tolerance, and presented a new capability system that (to the best of our knowledge) is the first to simultaneously address all three. Our evaluation shows microsecond-scale capability operations and low system overheads that remain constant regardless of data centre size. While transitioning cloud applications to capability-based security is non-trivial, this work seeks to reduce access control as a barrier to cloud-scale PoLA.

References

[1] [n. d.]. Machine Learning Service - Amazon Sagemaker AI - AWS. https://aws.amazon.com/sagemaker-ai/.
[2] [n. d.]. Vertex AI Studio. https://cloud.google.com/generative-ai-studio.
[3] Microsoft Azure. 2025. Azure OpenAI Service. https://azure.microsoft.com/en-us/products/ai-services/openai-service
[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* Morgan and Claypool Publishers.
[5] 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 *ACM SIGOPS Symposium on Operating Systems Principles.* https://doi.org/10.1145/1629575.1629579
[6] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus Van Der Merwe. 2017. CapNet: Security and Least Authority in a Capability-Enabled Cloud. In *Symposium on Cloud Computing.* https://doi.org/10.1145/3127479.3131209
[7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (Aug. 2013), 8:1–8:22. https://doi.org/10.1145/2491245

[8] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *International Conference on Management of Data*. https://doi.org/10.1145/2882903.2903741

[9] Aaron Elliott and S. Knight. 2010. Role Explosion: Acknowledging the Problem. In *Software Engineering Research and Practice*.

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3297858.3304013

[11] Simon Gerber. 2018. Authorization, Protection, and Allocation of Memory in a Large System. (2018).

[12] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems* 34, 1 (April 2016), 1–29. https://doi.org/10.1145/2893177

[13] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. SemperOS: A Distributed Capability System. In *USENIX Annual Technical Conference (ATC)*.

[14] Cheng Jin, Abhinav Srivastava, Yu Jin, and Zhi-Li Zhang. 2014. Secgras: Security Group Analysis as a Cloud Service. In *2014 IEEE 22nd International Conference on Network Protocols*. 215–220. https://doi.org/10.1109/ICNP.2014.42

[15] Cheng Jin, Abhinav Srivastava, and Zhi-Li Zhang. 2016. Understanding Security Group Usage in a Public IaaS Cloud. In *IEEE International Conference on Computer Communications (INFOCOM)*. https://doi.org/10.1109/INFOCOM.2016.7524508

[16] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. 2015. What You Need to Know About SDN Flow Tables. In *Intl. Conf. on Passive and Active Measurements (PAM)*.

[17] Henry M. Levy. 1984. *Capability-Based Computer Systems*. John Wiley & Sons.

[18] Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. 2003. *Capability Myths Demolished*. Technical Report SRL2003-02. John Hopkins University.

[19] S. J. Mullender and A. S. Tanenbaum. 1986. The Design of a Capability-Based Distributed Operating System. *Comput. J.* 29, 4 (Jan. 1986), 289–299. https://doi.org/10.1093/comjnl/29.4.289

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *USENIX Conference on Networked Systems Design and Implementation*.

[21] Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 150–163. https://doi.org/10.1109/CSF.2016.18

[22] J.H. Saltzer and M.D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (Sept. 1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[23] Matthew Sanders and Chuan Yue. 2017. Automated Least Privileges in Cloud-Based Web Services. In *ACM/IEEE on Hot Topics in Web Systems and Technologies*. https://doi.org/10.1145/3132465.3132470

[24] Matthew W Sanders and Chuan Yue. 2019. Mining Least Privilege Attribute Based Access Control Policies. In *Annual Computer Security Applications Conference*. https://doi.org/10.1145/3359789.3359805

[25] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. 2000. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control (RBAC '00)*. Association for Computing Machinery, New York, NY, USA, 47–63. https://doi.org/10.1145/344287.344301

[26] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Symp. on Operating Systems Principles (SOSP)*.

[27] Andrew S Tanenbaum, Sape J Mullender, and Robbert van Renesse. 1995. Using Sparse Capabilities in a Distributed Operating System. (1995).

[28] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3035918.3056101

[29] Lluis Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Hartig, and Mark Silberstein. 2022. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519569

[30] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symp. on Security and Privacy*.

[31] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symp. on Security and Privacy (SP)*.

[32] Jonathan Woodruff, Robert N. M. 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 *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2014.6853201

[33] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*.

[34] Ronghua Xu, Yu Chen, Erik Blasch, and Genshe Chen. 2018. BlendCAC: A BLockchain-Enabled Decentralized Capability-Based Access Control for IoTs. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. https://doi.org/10.1109/Cybermatics_2018.2018.00191

[35] Zhongyuan Xu and Scott D. Stoller. 2015. Mining Attribute-Based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing* 12, 5 (Sept. 2015), 533–545. https://doi.org/10.1109/TDSC.2014.2369048

[36] E. Yuan and J. Tong. 2005. Attributed Based Access Control (ABAC) for Web Services. In *IEEE International Conference on Web Services (ICWS'05)*. 569. https://doi.org/10.1109/ICWS.2005.25