MMCaps: Towards Fast and Secure IPC using MMU-Accelerated Memory Capabilities

Niklas Gollenstede
gollenstede@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Sören Tempel
tempel@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Lars Wrenger
wrenger@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Daniel Lohmann
lohmann@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Christian Dietrich
dietrich@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Abstract

Inter-process communication (IPC) is essential to securely exchange data between isolated processes, especially in highly-compartmentalized systems. However, established synchronous mechanisms are limited in message size and multi-core scalability; approaches based on shared memory suffer from either high synchronization overheads or *time-of-check to time-of-use (TOCTOU)* vulnerabilities.

We propose MMCaps, a new OS abstraction for fast and secure zero-copy asynchronous IPC. Conceptually, MMCaps are globally-unique capabilities to sharable buffers. Technically, MMCaps are provided by granting/revoking access to dedicated memory pages in the involved address spaces. The MMU implicitly enforces access checks on MMCap invocations in hardware, so no kernel mediation is needed.

Our Linux-based prototype shows that MMCaps outperform Linux pipes by 63 percent for four inflight messages and up to 221 percent for 64 inflight messages of 4 KiB.

CCS Concepts

• **Computer systems organization** → *Architectures*; • **Security and privacy** → **Operating systems security**.

Keywords

Operating System, Inter-Process Communication, Capability System, Memory Management Unit

ACM Reference Format:
Niklas Gollenstede, Sören Tempel, Lars Wrenger, Daniel Lohmann, and Christian Dietrich. 2026. MMCaps: Towards Fast and Secure IPC using MMU-Accelerated Memory Capabilities. In *19th European Workshop on Systems Security (EuroSec '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3803525.3804988

1 Introduction

IPC is an essential *operating system (OS)* mechanism for securely transmitting data between otherwise isolated processes. For highly compartmentalized systems (e.g., microkernels or Android), high message frequencies make IPC performance an essential property [23, 10, 17, 41]. In the past, synchronous fast-path IPC [23, 17] was the traditional choice. But *synchronous* IPC requires kernel activation and, in many cases, cross-core synchronization, limiting multi-core scalability [25, 41].

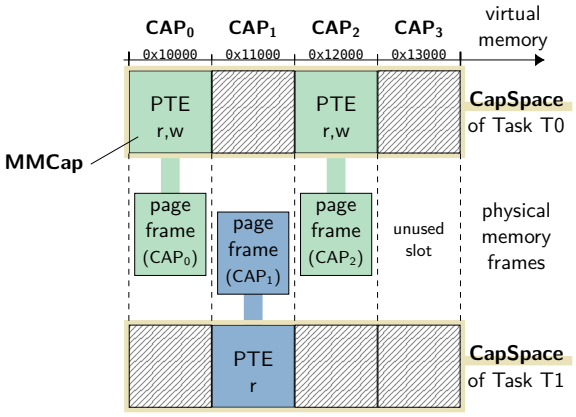


Figure 1: The CapSpace/MMCap Concept.

A viable alternative is asynchronous communication over shared-memory channels [3, 5]. This allows two processes to exchange messages without involving the kernel. However, because the sender retains access to the message buffer after transmission, it is easy to introduce TOCTOU vulnerabilities. These vulnerabilities can be prevented only by copying the message into the private memory of the receiver. Unfortunately, message copying has a significant impact on IPC performance [23, Sec. 5.2.3]. We believe that asynchronous IPC is the most viable approach for modern systems, but fine-grained access control is needed to enable secure zero-copy IPC operations.

Fortunately, capabilities [40, 26, 34] already provide a theoretical framework to express access control. IPC and capabilities are closely related: While IPC channels punch holes through the process isolation, capabilities can control those holes. For example, in Unix, a process needs a transmit capability, in the form of a writable file descriptor, to send a message over a pipe. Our idea

EuroSec '26, Edinburgh, Scotland Uk

https://doi.org/10.1145/3803525.3804988

is that messages themselves become specialized *memory-mapped capabilities* (MMCaps).

We envision a fast and secure IPC system built on MMCaps, which we present in this paper (see Fig. 1): Each address space contains a process-private CapSpace at a fixed virtual-memory address that maps all MMCaps the process has access to. An MMCap is the right to access a page frame and is owned by exactly one process. MMCaps can be moved between CapSpaces, but they always remain at the same offset, making pointers to MMCaps stable across processes.

The MMCap design is inspired by *single–address-space operating systems (SASOSs)* [27, 18, 6] as we share the logical structure of the CapSpace. This allows for lock-free synchronization in the kernel, enables *memory-management unit (MMU)*-accelerated data access from the user space, but avoids the risk of TOCTOU vulnerabilities caused by mutable payloads. In contrast to SASOSs, we keep processes isolated and do not require a new application model.

This work-in-progress paper claims these contributions:

- We describe the MMCap concept that allows for fast creation, access, and sharing of memory capabilities.
- We apply the MMCap concept to a Linux-based IPC implementation that protects against TOCTOU vulnerabilities while boosting performance through a zero-copy message exchange that is asynchronous and non-blocking.

2 Terminology

In this paper, we use the terminology established by Miller et al. [26]. A capability is a distinct object that grants a *subject* (the holder) the *authority* (specific access rights) to interact with a specific *resource*. If a subject wants to access a resource, it must *invoke* the corresponding capability. This invocation is mediated by a *security monitor*, which checks the validity of the capability and enforces its access rights. Additionally, a subject can *delegate* a capability to another subject, thereby transferring the authority. Capability systems implement different policies for delegation. Some systems allow unrestricted copying of capabilities [40, 22], while others restrict duplication through specific rights [34, 35] or even provide move semantics (e.g., Mach send-once ports [11]).

An MMCap is a specialized capability type that grants read or write access to a specific physical memory frame (resource). The invocation of an MMCap is done through memory operations (loads, stores). These invocations are checked by the MMU, which is our security monitor.

3 Mapped Memory Capabilities

The goal of our work is to provide a fast and secure OS abstraction for inter-process communication. The central building block of this abstraction is our MMCap primitive. In Sec. 3.1, we present the core concepts of this system and afterward apply them to IPC in Sec. 3.2.

The resulting system secures the inter-process communication payloads against TOCTOU vulnerabilities while boosting performance through a zero-copy message exchange that is asynchronous and non-blocking.

3.1 Core Concepts

The central element of our concept is the CapSpace (see Fig. 1), a shared virtual address range that is part of every *virtual address space (AS)* at a fixed location. It is divided into page-frame-sized *slots*, and we enforce three invariants: (1) Each task has its own private CapSpace with the same number of slots. (2) An MMCap is always mapped at the same CapSpace offset. (3) At any point in time, an MMCap is owned by exactly one task, and it is mapped in only that task's CapSpace. Together, these invariants ensure unique ownership of the MMCap at a fixed offset.

Tasks are the subjects of the capability system. Technically, a *page table entry (PTE)* in the CapSpace grants a task the authority to access the mapped physical page and thereby to the protected resource represented by this MMCap. Since the referenced physical page *is* the protected resource, a thread can invoke the authority by simply accessing the virtual memory (read, write, execute). Hence, the MMU acts as the security monitor, as it implicitly and transparently enforces checks on every access without any kernel involvement. The kernel is only activated in the error case of an access violation. An authority can be delegated by moving the MMCap from the CapSpace of one task to another. Technically, delegation is performed atomically by modifying the respective PTEs. This prevents TOCTOU vulnerabilities. Overall, our capability system exhibits the following properties:

(1) *Zero invocation overhead*. By using the MMU hardware as the security monitor, checking the access at every invocation involves no extra overhead. In particular, the kernel is not activated.
(2) *Stable pointers*. MMCaps keep their address across all address spaces, allowing them to be communicated and dereferenced across tasks.
(3) *Asynchronous delegation*. As MMCaps have unique slots, unusable by anything else, and are granted/revoked by just changing a PTE, they can be received without any synchronization with the receiver.
(4) *Explicitly bounded resources*. CapSpaces are preallocated by the kernel; senders cannot exhaust receivers' resources.

In the following, we leverage these properties to build a fast and secure OS abstraction for inter-process communication.

3.2 Inter-Process Communication

Based on our MMCap concept, we build an IPC system that achieves fast message transfer through remapped memory and – at the same time – secures payloads against TOCTOU vulnerabilities through memory capabilities. Fig. 2 illustrates the message exchange in the resulting IPC system. We decouple (a) the one-sided message transmission (MMCap delegation) from (b) the signaling mechanism. The left side of Fig. 2 shows how task T0 transmits a message to T1 via an MMCap CAP_0. On the right-hand side, T0 uses a non-blocking signaling FIFO to inform T1 about CAP_0's delivery.

Initially, T0 allocates the new CAP_0 in its CapSpace, and then writes a message `msg` to the underlying physical memory (i.e., the page frame) (①). Afterward, T0 transmits the message to T1 by moving the MMCap from its CapSpace to the CapSpace of T1 (②). This move is a combined unmap/map operation executed by the

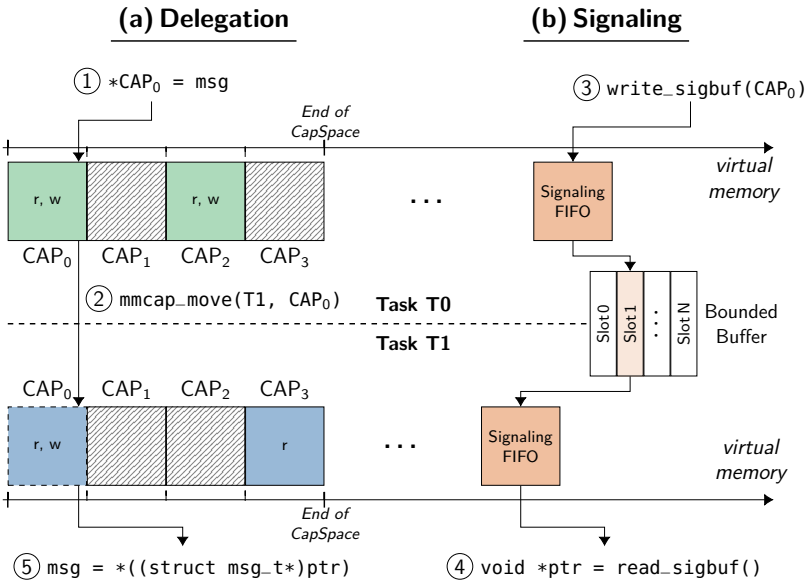


Figure 2: Utilization of our MMCap system for *zero-copy* IPC. The left side (a) illustrates *asynchronous* delegation of CAP_0 from task T0 to task T1. The right side (b) depicts how its availability can be signaled to T1 using a bounded, *non-blocking* buffer.

kernel, which ensures unique ownership (see Sec. 3.1). The uniqueness gives T1 the guarantee that CAP_0 cannot be modified by a third party (i.e., T0) when it invokes the received read authority (⑤). This eliminates the possibility of TOCTOU vulnerabilities. Additionally, authorities are checked by the MMU – not by the kernel – so no additional context switches are required.

In contrast to prior work [22], the message is transmitted asynchronously for the receiver – there is no rendezvous between T0 and T1 in the kernel that would require expensive cross-core signaling and synchronization of the receiver. While it would be trivial to build a synchronous mechanism on top, leaving signaling to the user provides for high flexibility regarding, for instance, batched message processing, event-based receivers, and different signaling methods (e.g., polling or batched signals).

In Fig. 2 (right side), we illustrate how this signaling *can* be implemented in an asynchronous manner. For this purpose, we use a dedicated non-blocking *first-in-first-out (FIFO)* signaling buffer between T0 and T1. This buffer is mapped in a shared memory region outside the CapSpace. It provides a fixed number of fixed-size slots, each slot large enough to store an MMCap reference (i.e., a pointer to the CapSpace). Writing to and reading from the buffer succeeds as long as the buffer is not full or empty, respectively, i.e., as long as the other end keeps up with processing the messages. In order to notify T1 of a new message, T0 writes the MMCap reference to the signaling buffer (③ in Fig. 2). As MMCap pointers are stable across all virtual address spaces, T1 reads the reference from the FIFO (④) and it can directly dereference it to obtain the message (⑤). Inverting the roles, T1 can reply to the message in the same way.

Summary. Based on MMCaps, we present a fast and secure IPC abstraction. The proposed design enables asynchronous, zero-copy message transfer through remapped memory and avoids TOCTOU vulnerabilities using MMCap's unique ownership guarantee.

4 Experiments with Linux

We are interested in the end-to-end performance of our MMCap-based IPC system and how it compares to existing mechanisms available in Linux. We chose Linux due to its mature implementation and the availability of different IPC channels to compare against. Furthermore, we also quantify the impact of *translation lookaside buffer (TLB)* management, since moving an MMCap to another CapSpace requires a PTE-unmap operation. For our experiments, we built an MMCap prototype system and implemented two benchmark scenarios, which we describe in the following sections.

4.1 The MMCap Prototype

Our prototype consists of two components: the MMCap kernel module that provides the memory-capability abstraction and the user-space signaling mechanism (see Fig. 2).

We built a kernel module for Linux 6.9.5 that provides the MMCap abstraction to the user space. Within the CapSpace address range, which is mapped at a fixed address in every process, we disable the Linux page-table modification primitives. Instead, we manipulate the page-table tree with lock-free operations, which we use to move 4 KiB or 2 MiB PTEs between CapSpaces. This is possible as the owner of an MMCap PTE also has implicit ownership of the corresponding slot in all other CapSpaces. For every move, we trigger Linux's existing TLB shootdown primitives to invalidate the old mapping. Our implementation is a modification of about 300 lines on top Morsels [15].

In the user space, we use a FIFO to signal the transmission of an MMCap. After the sender has moved an MMCap via the kernel interface to the receiver's CapSpace, depending on the scenario, it

inserts either a 16-byte routing header or an 8-byte pointer referencing the MMCap into the shared-memory FIFO. We use a lock-free FIFO implementation with atomic read and write indices, and keep data copies to a minimum (8 or 16 bytes). Receivers read at least one word from the MMCap to cause a capability access to the referenced page frame.

Hardware Setup. We run all experiments on a 64-core AMD EPYC 9554P (without SMT or boosting) at 3.1 GHz with 12×32 GiB DDR5 4800, with NixOS/nixpkgs 25.11 (@ 2025-12-26) as user space and for all tooling, including clang 21.1.2. To prevent our results from being dominated by scheduler overhead, we limit the total number of threads to a maximum of 64, ensuring each thread runs on its own core.

4.2 IPC Throughput

As described in Sec. 3, our IPC system is tailored to high throughput by facilitating a zero-copy message exchange. In this section, we use end-to-end benchmarks to compare the achieved throughput with established IPC primitives provided by Linux.

For this purpose, we implemented a centralized message broker process in user space. The broker is responsible for mediating between several IPC clients, which cannot communicate directly with each other. Instead, all clients are connected to the broker through private IPC channels on which they send and receive messages. The broker forwards the message to the addressed communication partner and informs the sender about the delivery.

We use this scenario as it is a common communication pattern for inter-process communication. Notably, the DBus [30] protocol – a central building block of the Linux desktop ecosystem – makes use of such a centralized IPC architecture. In the desktop context, large messages are often transmitted at a high frequency, which makes performance of the underlying IPC primitive a vital optimization goal.

We implement the outlined communication topology on top of different IPC mechanisms. Specifically, we facilitate our MMCap-based IPC system (including the FIFO-based signaling outlined in Sec. 3.2) and compare it against existing Linux IPC mechanisms: pipes, Unix domain sockets, and POSIX message queues. Apart from the IPC primitive, the setup is parametrized over a variable number C of clients, the size of the message payload S, and the maximal number of messages L each client keeps in flight before receiving a reply from the broker. The routing requires small (16-byte) headers to communicate message types, IDs, destinations and sources. For MMCap-based IPC, we use the IDs as MMCap indices and include these headers in the signaling FIFO (see Sec. 3.2), leaving the MMCaps themselves entirely for the payload. With the other IPC mechanisms, we prepend the header to the message payload.

For our evaluation, the clients continuously send messages to each other. We measure the throughput in terms of forwarded packets at the broker. One benchmark run measures eight consecutive one-second intervals (after two seconds of warm-up); we repeat each run 10 times and report the mean and the standard deviation of the 80 one-second measurements. The results over different amounts of clients, payload sizes, and inflight limits are shown in Fig. 3.

Fig. 3 (a)-(c) show the message throughput of the shared-memory FIFO, full MMCap IPC, and Unix pipes over a varying amount of clients. In facet (a), we see that in low concurrency situations (e.g., for $L \equiv 1$), the FIFO slots are not used to their full capacity; thus, the FIFO imposes a performance ceiling on the MMCap-based IPC. However, our system is tailored to highly concurrent situations (many inflight messages and many communication partners). For large L and C, we see that the shared-memory FIFO is a well-suited signaling mechanism for MMCap-based IPC.

Fig. 3 (b) and (c) compare the throughput for 4 KiB page-sized message payloads. We show this point in the S-dimension, as it is the natural granularity where all IPC mechanisms *could* benefit from transparent zero-copy techniques. At $L \equiv 1$, the performance of MMCap-based IPC and pipes is on par as the signaling FIFO limits the MMCaps.

Our approach outperforms Linux pipes for $L > 1$, becoming up to 63 percent faster for four inflight messages and up to 221 percent for 64 inflight messages. This illustrates that, for highly-concurrent messaging, our asynchronous IPC approach significantly outperforms established, synchronous IPC primitives supported by Linux.

For the throughput comparison, we focused on Linux pipes, as they are the fastest of the existing IPC primitives in our measurements. To further contextualize the results, we show the forwarding rates of additional IPC mechanisms in Fig. 3 (d). Here, we focus on the high-throughput case with $L \equiv 64$ and $C \equiv 64$. For small message sizes, pipes are the fastest IPC mechanism, outperforming sockets (stream and datagram) as well as POSIX message queues by a factor of 2.6–3.5 for $S \equiv 32$ bytes. At this small message size, pipes are 9.1 percent faster than MMCaps. At around 512 bytes, MMCaps outperform pipes as an IPC primitive. And, as message sizes approach 4 KiB, the lead of pipes over the other IPC mechanisms disappears.[1]

From 4 KiB to 2 MiB, the transfer rate keeps decreasing, reaching only 819 transfers per second for pipes at $S \equiv 2$ MiB, one-259th of pipes at 4 KiB. Even though we use page-aligned buffers for all sends and receives, Linux clearly performs copies for all existing IPC primitives. In contrast, our approach facilitates a zero-copy message exchange through remapped memory and thereby – relative to the page-size, as we also support 2 MiB huge pages – achieves a constant high-throughput forward rate in Fig. 3 (d).

Summary. The results show that our proposed IPC approach achieves high throughput and outperforms established, synchronous IPC primitives by up to 221 percent for high concurrency situations ($L \equiv 64$ and $C \equiv 64$).

4.3 MMCaps and TLB Management

Our MMCap IPC implementation frequently maps and unmaps pages in the CapSpaces of different processes. A well-known challenge with so many virtual memory operations is TLB management [29, 2]. The TLB is an explicitly-managed per-core hardware cache of recently used PTEs that greatly accelerates memory accesses. When pages are unmapped, the OS has to invalidate TLB entries on all cores that *may* have cached the previous PTE. This

[1] Note that sockets and POSIX message queues have a system-wide upper limit on the message size and thus we cannot provide measurements for all message sizes for these primitives.

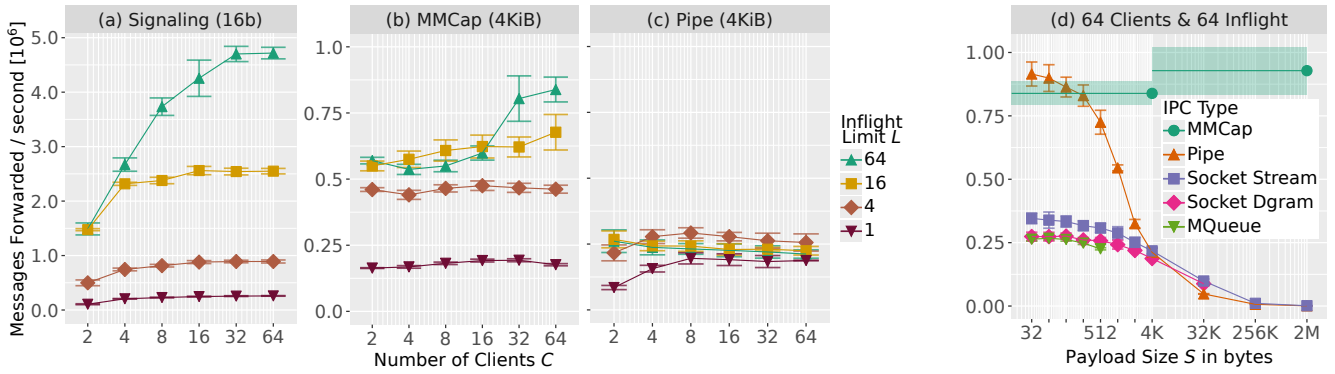
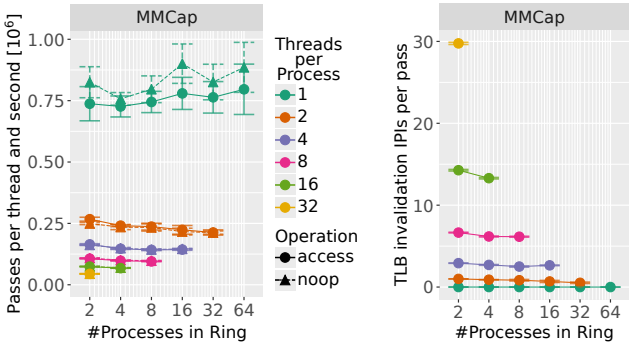


Figure 3: Message forwarding rate of the server in the router scenario.



(a) Multi-Threaded Passing Rates **(b) TLB Invalidation IPIs**

Figure 4: Multi-core scalability in the parallel ring setup: Parallel multithreading causes the same AS to be used on multiple cores, requiring TLB invalidation IPIs.

becomes expensive when multiple threads of the same process run on different cores, as the TLB invalidations then require costly *inter-processor interrupts (IPIs)*. For us, only the sender process of an MMCap must invalidate its TLB when the MMCap is moved away. Due to MMCap's uniqueness, the receiver process can never have a valid TLB entry at the MMCap's slot.

To quantify these TLB invalidation costs, we use a second benchmark scenario with P processes and T threads per process. We form T independent rings, each connecting one thread from every process in a circular chain. In these rings, we circle 4 KiB MMCaps as fast as possible. We vary (P, T) from $(64, 1)$ to $(2, 32)$, keeping the total thread count $P \cdot T$ at or below 64 (= number of CPU cores).

We use a per-thread inflight limit of 64, i.e., each thread can send up to 64 MMCaps before receiving an MMCap from its predecessor. The benchmark is therefore not limited by the signaling (upper bound of 5M average passes per second).

We measured two cases: (1) a *noop* case, where the threads only receive and immediately send the MMCaps without accessing them, and (2) an *access* case, where the threads increment an integer stored in the MMCaps before sending them on. Even though only the latter case actually accesses the MMCap and thus is expected to load TLB entries, the kernel can not know this and still invalidates TLB entries on every send even in the noop case. The difference between the two cases therefore quantifies the cost of accessing the memory and filling the TLB entries.

In Fig. 4a we report the number of MMCaps passed through the first process. For one thread per process ($T = 1$), our maximum MMCap passing rate of $8.86 \cdot 10^5$ is similar to that of the router with the same configuration ($C = 64$, $L = 64$). In this single-threaded case, actually accessing the MMCap reduces the passing rate by 8.9 %.

Generally, if we keep the number of threads per process (T) stable, the passing rate per thread decreases only slightly with the number of processes (P). If, however, we increase the number of threads in the same process, then passing performance decreases significantly. With $P = 2$, the rate decreases by 64 % if we increase T from 1 to 2 and by another 38 % from 2 to 4.

We expect that the origin of this performance degradation is the required cross-core TLB invalidation, caused by multiple threads sharing the same AS. To confirm this hypothesis, we record the number of TLB-invalidation IPIs the system performs while executing our benchmark. In Fig. 4b, we show the number of IPIs per passed MMCap. With $T = 1$, the OS can avoid (almost) all IPIs as only the current CPU can hold relevant TLB entries. Starting at $T = 2$, we see that the number of IPIs linearly increases with the number of threads, as every pass needs to invalidate a TLB entry on each other core that executes a thread of the same process. In the worst case, with $T = 32$, about 29.8 IPIs are sent per MMCap pass, meaning that nearly half of the cores need to invalidate a TLB entry for each MMCap sent.

Summary. Our proposed CapSpace concept, where a received MMCap automatically becomes visible to all threads of the process, suffers from a large number of TLB-shootdown IPIs.

5 Future Work

Our MMCap prototype is still in its early stages. In the following, we outline several possible future directions and improvements.

Remote TLB invalidations. The high cost of TLB shootdowns in multithreaded scenarios motivates future optimizations. Already, hardware support for more efficient remote TLB invalidations, such as AMD's `INVLPGB` instruction [32] or Intel's RAR [19], is on the horizon, which may alleviate some of the observed overheads. Even with such hardware support, however, cross-core TLB invalidations

will likely remain a bottleneck. Also, even better for MMCap would be a hardware mechanism to re-tag TLB entries with another AS identifier when moving them between CapSpaces. This would directly make the TLB entry available in the target, preventing the following TLB miss.

Thread-local CapSpace. Another way to avoid cross-core TLB invalidations would be to have thread-local CapSpaces instead of process-wide ones. This would, however, require explicit intra-process passing to transfer MMCaps between threads. Additionally, the TLB itself might be negatively affected: currently, all threads of a process share the same AS identifier, which is used to tag TLB entries and cache them across context switches. With thread-local CapSpaces, threads would need to have different AS identifiers as well, which may reduce the effective TLB size per thread. There is also a hardware limit on the number of AS identifiers (4096 on x86), which might be exhausted more quickly. However, both effects only seem to be relevant for extremely multithreaded scenarios [8, 1, 33, 13].

Peripheral devices. Aside from TLB management, we also want to integrate MMCaps with DMA-devices, like SSDs or network cards. Conceptually, our asynchronous zero-copy mechanism is entirely compatible with the IOMMU, making devices first-class MMCap communication partners, with the same security benefits. Still, we need support for MMCaps in device drivers and IOMMU/IOTLB management, which we leave for future work.

Confused-deputy attacks. Another limitation of our current prototype is that we do not yet mitigate *confused-deputy attacks* [26]. Because signaling is left to user space, confused-deputy attacks are possible; that is, an attacker can claim to have been the sender of a received MMCap. To prevent this, the kernel could store the last sender of each MMCap, which is unambiguously possible due to MMCap's unique ownership model, and provide this information to the user space, for example, by a shared read-only mapping. Our current prototype does not yet implement this functionality, which would make it easily possible to retrieve the actual sender of an MMCap.

Microkernel integration. Eventually, we want to integrate MMCaps into a microkernel-based OS, which benefits most from secure and efficient IPC. For this preliminary work, we chose Linux as a widely used and mature OS that includes various IPC mechanisms to compare against.

6 Related Work

Due to its importance for microkernels, there is a vast body of prior work on inter-process communication. In the following, we outline related work regarding capability operating systems, IPC performance, and tagged-memory architectures.

6.1 Capability Operating Systems

Capabilities were pioneered by HYDRA [40] and have been regularly featured in research and security-focused OSes [16, 34, 21, 22]. For example, Capsicum [39] provides a POSIX-compatible layer for FreeBSD, and, with Fuchsia/Zircon [28], Google is actively working on a capability-based OS. With microkernels, capabilities and IPC are traditionally close topics: For example, in the L4Re (L4/Fiasco), every capability is a synchronous IPC gate to the kernel or another task, and supports "capability invocations" [22, 31]. In contrast, MMCaps can be passed asynchronously and aim to address the reported scalability issues [25] of L4Re.

6.2 IPC Performance

Given the importance of IPC performance, a large body of related work concerns itself with developing optimizations. As software-only approaches [4, 23, 12] are inherently limited by required context switches, prior work also investigates the utilization of hardware features [38, 24, 41, 9]. Unfortunately, the majority of prior approaches require custom or emerging hardware features with limited availability [38, 41, 9]. SkyBridge [24] uses virtualization features of commodity hardware but – contrary to MMCap– retains a synchronous IPC model.

6.3 Tagged-Memory Architectures

In recent years, there has generally been an increased interest in hardware-enforced capabilities that build upon tagged-memory architectures [20]. These works include CHERI [7], Capstone [42], and CODOM [37]. CODOM was also used to build an IPC system [38]. However, tagged-memory architectures are not yet available as commodity hardware and require custom CPU instructions, thus hindering widespread adoption. Further, a CHERI capability (the most prominent approach) protects a virtual-address range (resource), which provokes an aliasing problem when capabilities are shared between address spaces. In contrast, MMCaps protect a physical frame, whose address we keep stable across address spaces.

7 Conclusion

In this work-in-progress paper, we present MMCaps: a capability system tailored to fast and secure IPC. The proposed design achieves fast message transfer by facilitating asynchronous, zero-copy communication through remapped memory. The communication is secured against TOCTOU vulnerabilities on payloads because each MMCap can only be mapped in one process' CapSpaces at a time (unique ownership). Experiments conducted with a Linux-based prototype implementation illustrate that our proposed approach outperforms established IPC primitives – in terms of forwarded messages – by up to 221 percent. We further evaluated the impact of cross-core TLB invalidations, induced by the unique ownership property. For future work, we aim to mitigate the impact of TLB shootdowns by experimenting with thread-local CapSpaces [36]. Considering the promising throughput performance results, we also plan to integrate our MMCap implementation with a microkernel, where IPC performance is paramount. We publish a pre-compiled artifact of our evaluation along with its user-space code with this paper [14]. The release of the kernel code is awaiting related publications, once published the code will be made available and the artifact will be updated accordingly.

8 Acknowledgments

We thank our anonymous reviewers for their helpful and constructive comments. This work was partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '20). Association for Computing Machinery, Lausanne, Switzerland, 283–300. ISBN: 9781450371025. DOI: 10.1145/3373376.3378468.

[2] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems* (EuroSys'20), 1–14. DOI: 10.1145/3342195.3387518.

[3] 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 22nd ACM Symposium on Operating Systems Principles* (SOSP '09) (Big Sky, Montana, USA). ACM Press, New York, NY, USA, 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579.

[4] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. 1989. Lightweight remote procedure call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (SOSP '89). Association for Computing Machinery, New York, NY, USA, 102–113. ISBN: 0897913388. DOI: 10.1145/74850.74861.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9, 2, (May 1991), 175–198. DOI: 10.1145/103720.114701.

[6] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. 1992. Opal: a single address space system for 64-bit architecture address space. In *Third Workshop on Workstation Operating Systems*, 80–85. DOI: 10.1109/WWOS.1992.275684.

[7] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. 2015. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '15). ACM New York, NY, USA. DOI: 10.1145/2694344.2694367.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. Radixvm: scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (EuroSys '13). Association for Computing Machinery, Prague, Czech Republic, 211–224. ISBN: 9781450319942. DOI: 10.1145/2465351.2465373.

[9] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: cryptographically-enforced in-process capabilities for modern arm architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (CCS '23). Association for Computing Machinery, Copenhagen, Denmark, 874–888. ISBN: 9798400700507. DOI: 10.1145/3576915.3623079.

[10] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (SOSP'95) (Copper Mountain, CO, USA). ACM Press, New York, NY, USA, (Dec. 1995), 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224076.

[11] Free Software Foundation. 2008. *The GNU Mach Reference Manual*. (0.4 ed.). For GNU Mach version 1.3.99. Free Software Foundation. (Nov. 2008). https://www.gnu.org/software/hurd/gnumach-doc/.

[12] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings Operating System Design and Implementation* (OSDI '99). Usenix Association. New Orleans, LA, USA, (Oct. 1999), 87–100. https://www.usenix.org/legacy/events/osdi99/gamsa.html.

[13] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. 2024. Scalable and effective page-table and TLB management on NUMA systems. In *2024 USENIX Annual Technical Conference* (USENIX ATC '24). USENIX Association, Santa Clara, CA, (July 2024), 445–461. ISBN: 978-1-939133-41-0. https://www.usenix.org/conference/atc24/presentation/gao-bin-scalable.

[14] Niklas Gollenstede. 2026. MMCap EuroSec26 artifact. (Mar. 2026). DOI: 10.5281/zenodo.19065032.

[15] Alexander Halbuer, Christian Dietrich, Florian Rommel, and Daniel Lohmann. 2023. Morsels: explicit virtual memory objects. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (DIMES '23). Association for Computing Machinery, Koblenz, Germany, (Oct. 2023). DOI: 10.1145/3609308.3625267.

[16] Norman Hardy. 1985. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19, 4, 8–25. DOI: 10.1145/858336.858337.

[17] Gernot Heiser and Kevin Elphinstone. 2016. L4 microkernels: the lessons from 20 years of research and deployment. *ACM TOCS*, 34, 1, (Apr. 2016). DOI: 10.1145/2893177.

[18] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russel, and Jochen Liedtke. 1998. The Mungi single-address-space operating system. *Software: Practice and Experience*, 18, 9, (July 1998).

[19] Intel Corporation. 2021. Remote Action Request: White Paper. White Paper 341431-001US. Version Revision 1.0. Intel Corporation, (July 2021). Retrieved July 23, 2025 from https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf.

[20] Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi. 2022. TAG: tagged architecture guide. *ACM Comput. Surv.*, 55, 6, (Dec. 2022). DOI: 10.1145/3533704.

[21] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (SOSP '09) (Big Sky, Montana, USA). ACM Press, New York, NY, USA, 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[22] Adam Lackorzynski and Alexander Warg. 2009. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (EuroSys '09). ACM, Nuremburg Germany, (Mar. 31, 2009), 25–30. ISBN: 978-1-60558-464-5. DOI: 10.1145/1519130.1519135.

[23] Jochen Liedtke. 1993. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM Press. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168633.

[24] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019* (EuroSys '19). Association for Computing Machinery, Dresden, Germany. ISBN: 9781450362818. DOI: 10.1145/3302424.3303946.

[25] Till Miemietz, Matthias Hille, Viktor Reusch, Lars Wrenger, Jana Eisoldt, Jan Klötzke, Max Kurze, Adam Lackorzynski, Michael Roitzsch, and Hermann Härtig. 2025. MettEagle: costs and benefits of implementing containers on microkernels. In *19th Symposium on Operating System Design and Implementation* (OSDI '25). Boston, MA, (July 2025). https://www.usenix.org/conference/osdi25/presentation/miemietz.

[26] Mark S Miller, Ka-Ping Yee, and Jonathan Shapiro. 2003. Capability myths demolished. Tech. rep.

[27] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. 1993. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the USENIX Symposium on Microkernels and other Kernel Architectures*, 31–43. https://www.usenix.org/legacy/publications/library/proceedings/micro93/murray.html.

[28] Francesco Pagano, Luca Verderame, and Alessio Merlo. 2021. Understanding fuchsia security. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 12, 3, (Sept. 2021), 47–64. DOI: https://doi.org/10.22667/JOWUA.2021.09.30.047.

[29] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '18). Association for Computing Machinery, Williamsburg, VA, USA, 679–692. ISBN: 9781450349116. DOI: 10.1145/3173162.3173203.

[30] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, and David Zeuthen. 2024. D-Bus specification. Version 0.43. (Oct. 29, 2024). Retrieved Feb. 2, 2026 from https://dbus.freedesktop.org/doc/dbus-specification.html.

[31] L4Re Project. 2025. L4 inter-process communication (IPC). L4Re Operating System Framework documentation. (May 2025). Retrieved July 10, 2025 from https://l4re.org/doc/l4re_concepts_ipc.html.

[32] Adam Riel. 2025. LWN: AMD broadcast TLB invalidation. (Feb. 2025). Retrieved Oct. 29, 2025 from https://lwn.net/Articles/1008065/.

[33] Florian Rommel, Christian Dietrich, Birte Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. 2020. From global to local quiescence: wait-free code patching of multi-threaded processes. In *14th Symposium on Operating System Design and Implementation (OSDI '20)*. (Nov. 2020), 651–666. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/rommel.

[34] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (ACM SIGOPS Operating Systems Review) (Charleston, South Carolina, USA). ACM Press, New York, NY, USA, 170–185. ISBN: 1-58113-140-2. DOI: 10.1145/319344.319163.

[35] Stephen Smalley and James Carter. 2018. Security in zephyr and fuchsia. In *Linux Security Summit North America*. Vancouver, Canada.

[36] Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. 2023. MELF: multivariant executables for a heterogeneous world. In *2023 USENIX Annual Technical Conference (USENIX '23)*. USENIX Association, Boston, MA, (July 2023), 257–273. ISBN: 978-1-939133-35-9. https://www.usenix.org/conference/atc23/presentation/tollner.

[37] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (ISCA '14). IEEE Press, Minneapolis, Minnesota, USA, 469–480. ISBN: 9781479943944. DOI: 10.1145/2678373.2665741.

[38] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2017. Direct inter-process communication (dIPC): repurposing the CODOMs architecture to accelerate IPC. In *Proceedings of the Twelfth European Conference on Computer Systems* (EuroSys '17). Association for Computing Machinery, Belgrade, Serbia, 16–31. ISBN: 9781450349383. DOI: 10.1145/3064176.3064197.

[39] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*. USENIX Association, Washington, DC, (Aug. 2010). https://www.usenix.org/conference/usenixsecurity10/capsicum-practical-capabilities-unix.

[40] William Allan Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, Charles Pierson, and Fred J. Pollack. 1974. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17, 6, (June 1974), 337–345. DOI: 10.1145/355616.364017.

[41] Yubin Xia, Dong Du, Zhichao Hua, Binyu Zang, Haibo Chen, and Haibing Guan. 2022. Boosting inter-process communication with architectural support. *ACM Trans. Comput. Syst.*, 39, 1–4, (July 2022). DOI: 10.1145/3532861.

[42] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: a capability-based foundation for trustless secure memory access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, (Aug. 2023), 787–804. ISBN: 978-1-939133-37-3. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason.