Capability-Based Security for Systems Programming

Unforgeable capability tokens in LateralusOS

Lateralus Language

bad-antics · January 2024 · Lateralus Language Research

ABSTRACT We describe the capability-based security model in LateralusOS, where access to files, network sockets, and hardware is mediated by unforgeable capability tokens. We show how capability passing composes with Lateralus pipeline operators, enabling least-privilege security policies that are enforced at compile time rather than runtime. We compare our approach to POSIX discretionary access control, Linux mandatory access control via SELinux and AppArmor, and prior capability systems including EROS, L4, Capsicum, and Google Fuchsia. Micro-benchmark results show a median capability-check overhead of 38 ns on an RV64GC core running at 1 GHz, and we demonstrate that the confused deputy problem is structurally impossible under the capability model without any programmer annotation burden.

1. Motivation

Modern operating systems grant permissions based on the identity of the process requesting an operation, not the specific code path that triggered the request. A setuid helper binary that compiles user-submitted code, for instance, may be tricked into writing to files the calling user does not own simply because the kernel checks the process UID and finds it sufficient. This is the confused deputy problem, named by Hardy (1988): a privileged service acts as a confused deputy when it exercises authority it holds on behalf of an attacker-controlled input.

Capability-based security resolves the confusion by making the authority explicit in the code path itself. Instead of checking the caller's UID at the kernel boundary, every resource access requires the caller to present a token — a capability — that the kernel issued when the resource was created. Without the token, no access is possible, regardless of the caller's identity. The token itself carries only the rights the issuer chose to include, enabling fine-grained least-privilege by construction.

LateralusOS makes capability tokens a first-class feature of the type system. The Cap<T> type in the Lateralus standard library is a linear type: it cannot be copied or aliased, only moved or explicitly delegated. The kernel validates capability tokens at syscall boundaries using a handle table indexed by a 64-bit unforgeable token, and the Lateralus compiler enforces linearity statically, so a program that compiles is guaranteed never to use a revoked or delegated-away capability.

This paper is organized as follows. Section 2 surveys prior capability systems. Section 3 introduces the Lateralus capability model and the Cap<T> type. Sections 4 through 8 cover capability lifecycle operations. Sections 9 through 12 describe domain-specific capabilities for files, sockets, and hardware. Section 13 describes kernel enforcement. Sections 14 through 16 compare the model to POSIX, DAC, and MAC. Section 17 presents benchmarks, and Sections 18 through 20 discuss the attack surface, limitations, and future work.

2. History of Capability Systems

Capability-based security was proposed by Dennis and Van Horn in 1966 as part of the Multics design. The central idea — that an object reference and the rights to use it should be inseparable — was radical at a time when protection was implemented as access-control lists maintained by the OS. The original capability model was never fully realized in Multics, partly because hardware support was limited.

The EROS operating system (Shapiro et al., 1999) provided a pure capability kernel running on IA-32. Every process started with a fixed set of capabilities and could only acquire new ones via explicit delegation. EROS provided process-as-capability by representing every IPC endpoint as a capability, so inter-process communication required holding the peer's endpoint cap. EROS demonstrated that a full, orthogonally persistent capability OS was practical on commodity hardware.

L4 (Liedtke, 1995) took a different angle: a minimal microkernel where the kernel provided only threads, address spaces, and IPC. Capability-flavored access control was layered on top. seL4 (Klein et al., 2009) formalized L4's capability model and produced a machine-checked proof of functional correctness, making it the first OS kernel with a full formal proof of security invariants.

Capsicum (Watson et al., 2010) retrofitted capability-based sandboxing into FreeBSD 9. A process calls cap_enter() to enter capability mode, after which every file-descriptor operation requires a capability right bitmask associated with the fd. Capsicum is notable for requiring minimal changes to existing UNIX programs and for its adoption in the base FreeBSD system.

Google Fuchsia uses Zircon, a microkernel with a capability model based on handles. Every kernel object (VMO, channel, port, process, job) is accessed via a handle that encodes a rights bitmask. Handles cannot be forged; they exist in a per-process handle table and are explicitly transferred between processes via channels. LateralusOS draws heavily on Fuchsia's handle model but adds compile-time linearity enforcement, which Fuchsia does not attempt.

3. The Lateralus Capability Model

A capability in LateralusOS is a 128-bit token consisting of a 64-bit kernel object identifier and a 64-bit rights bitfield. The kernel maintains a global capability table mapping object IDs to reference-counted resource descriptors. The rights field encodes up to 64 distinct operations; currently 18 rights are defined across the file, socket, and hardware domains.

```
// Rights bitfield layout (18 of 64 bits allocated)
const RIGHT_READ:       u64 = 1 << 0;
const RIGHT_WRITE:      u64 = 1 << 1;
const RIGHT_EXEC:       u64 = 1 << 2;
const RIGHT_MMAP:       u64 = 1 << 3;
const RIGHT_SEEK:       u64 = 1 << 4;
const RIGHT_STAT:       u64 = 1 << 5;
const RIGHT_TRUNCATE:   u64 = 1 << 6;
const RIGHT_CONNECT:    u64 = 1 << 7;
const RIGHT_ACCEPT:     u64 = 1 << 8;
const RIGHT_SEND:       u64 = 1 << 9;
const RIGHT_RECV:       u64 = 1 << 10;
const RIGHT_BIND:       u64 = 1 << 11;
const RIGHT_DMA_MAP:    u64 = 1 << 12;
const RIGHT_IRQ_MASK:   u64 = 1 << 13;
const RIGHT_IOPORT:     u64 = 1 << 14;
const RIGHT_DELEGATE:   u64 = 1 << 15;
const RIGHT_REVOKE:     u64 = 1 << 16;
const RIGHT_INSPECT:    u64 = 1 << 17;
```

The Lateralus type Cap<T> wraps a token and statically records the resource kind T (for example FileResource or TcpSocket). The type is **linear**: the compiler tracks each capability binding and emits an error if a capability is used after it has been moved, delegated, or implicitly dropped. Dropping a capability decrements the kernel refcount; when the refcount reaches zero the resource is freed.

```
// Capability type declaration in the Lateralus standard library
linear type Cap<T> {
    token:  u128,      // kernel-issued unforgeable handle
    rights: u64,       // subset of rights granted at creation
    _phantom: PhantomData<T>,
}

// Linearity means: each Cap<T> can be used exactly once
fn example() {
```

```
    let cap = fs::open("/etc/passwd", RIGHT_READ); // Cap<FileResource>
    let data = cap |> fs::read_all;  // cap moved here
    // cap |> fs::read_all;  // compile error: use after move
}
```

The kernel enforces the token at every syscall involving a resource. It looks up the object ID in the capability table, checks that the rights field covers the requested operation, and verifies that the token value matches the record in the table. If any check fails, the syscall returns Err(CapError::Denied) — no privilege escalation is possible because the kernel never grants permissions that were not in the original capability.

4. The Cap Type in Detail

Creating a capability requires a source of authority. LateralusOS provides three root capabilities to a process at startup: cap_fs_root for filesystem access, cap_net_root for network access, and cap_hw_root for hardware access. Each root capability has the full rights bitfield set. The process must narrow these into specific capabilities before using them.

```
// Lateralus startup: three root capabilities are delivered
// by the kernel via a special init message.
fn main(caps: StartupCaps) {
    let StartupCaps { fs, net, hw } = caps;
    // fs: Cap<FsRoot>   - full filesystem authority
    // net: Cap<NetRoot> - full network authority
    // hw: Cap<HwRoot>   - full hardware authority

    // Narrow fs to read-only access to /var/log
    let log_cap = fs |> cap::restrict("/var/log", RIGHT_READ | RIGHT_STAT);
    spawn_log_reader(log_cap); // move cap to child task
}
```

The cap::restrict() function is implemented as a kernel call that creates a new capability token with a subset of the parent's rights and a path constraint. The parent token is consumed (moved) by the call, and the returned child token carries strictly fewer rights. The kernel enforces this monotonic narrowing property: it is impossible to produce a capability with more rights than the token used to create it.

```
Monotonic Narrowing:
  If cap_child = restrict(cap_parent, path, rights_mask),
  then rights(cap_child) ⊆ rights(cap_parent)
  and  scope(cap_child) ⊆ scope(cap_parent).
```

The Cap<T> type carries a phantom type parameter T that records the resource kind at the Lateralus type level. This means passing a Cap<TcpSocket> where a Cap<FileResource> is expected is a compile-time type error, even if the rights bits would permit the operation. The type system acts as a first line of defense; the kernel token check is a second, independent enforcement layer.

The DROP implementation for Cap<T> issues a cap_close syscall, which decrements the reference count on the kernel object and, if the count reaches zero, immediately frees the resource. Because the type is linear, the compiler statically guarantees that cap_close is called exactly once per capability. This eliminates both capability leaks (open file descriptors that are never closed) and double-close bugs that plague POSIX fd-based code.

5. Unforgeable Tokens via Linear Types

The security guarantee of capability tokens rests on two properties: unforgeability and non-duplicability. Unforgeability means that a process cannot construct a valid token by guessing or arithmetic; non-duplicability means a process cannot copy an existing token to share it with an untrusted peer without going through a kernel-mediated delegation operation.

Unforgeability is achieved at the hardware level. LateralusOS runs on RV64GC with physical memory protection (PMP) regions that prevent userspace from reading the kernel capability table. A token value observed in userspace registers cannot be looked up in the kernel table directly; it must be presented via a syscall, at which point the kernel verifies the complete 128-bit value against the table. The table is populated only via sanctioned creation paths (cap_create, cap_restrict); there is no cap_forge syscall.

```
// Syscall ABI (assembly stub, RV64)
// Syscall number in a7; args in a0..a5; token in (a0, a1) as u128.
cap_check:
    li   a7, SYSCALL_CAP_CHECK
    ecall                    // trap to kernel
    // kernel validates token[127:64] in cap_table[a0]
    // kernel checks rights[a2] ⊆ cap_table[a0].rights
    ret
```

Non-duplicability is enforced by the Lateralus borrow checker. The Cap<T> type does not implement the Copy trait. Any attempt to bind a capability to two names simultaneously is rejected at compile time with error E0509: capability used after move. The only way to share access is via cap::delegate(), which is a kernel operation that creates a child token and revokes the parent.

Together these properties mean that the set of valid capabilities at runtime is exactly the set of tokens that the kernel issued and that have not yet been revoked or dropped. No amount of computation in userspace can add to this set. The security perimeter is therefore the kernel capability table, not the correctness of any userspace policy code.

6. Capability Creation and Delegation

New resource capabilities are created by narrowing an existing capability using cap::open() for files and cap::bind() for sockets. Both operations consume the root capability and return a narrowed one, making the call provably origin-traceable: every capability in the system has a lineage that terminates at one of the three startup roots.

```
// Creating a file capability from the fs root
fn open_config(fs: Cap<FsRoot>) -> Result<Cap<FileResource>, CapError> {
    fs |> cap::open("/etc/app.conf", RIGHT_READ | RIGHT_STAT)
}

// Creating a TCP listen socket capability
fn listen_8080(net: Cap<NetRoot>) -> Result<Cap<TcpListener>, CapError> {
    net |> cap::bind_tcp("0.0.0.0:8080", RIGHT_ACCEPT | RIGHT_RECV | RIGHT_SEND)
}
```

Delegation transfers a capability from one task to another. The delegating task calls cap::delegate(cap, target_task_id), which sends the token to the kernel, which removes it from the delegator's handle table and inserts it into the target's. The delegator's capability is consumed in the process; after delegation, the delegating task has no access to the resource.

```
// Delegating a read-only file cap to a child task
fn spawn_worker(cap: Cap<FileResource>, child: TaskId) {
    let child_cap = cap::delegate(cap, child); // cap moved, child_cap is the receipt
```

```
    // cap is no longer valid here; child task now holds it
    child.send(WorkerMsg::StartWithCap(child_cap));
}
```

Partial delegation is also supported: the delegator may further restrict the rights before transfer using cap::restrict() followed by cap::delegate(). A common pattern is for a parent task to hold a read-write file capability and delegate only a read-only capability to a worker that processes the file. The compiler enforces that the parent cannot retain the original read-write capability after the restricted copy is produced, because the restriction call consumes the parent token.

Capabilities may also be split into a set of rights-partitioned sub-capabilities using cap::split(). This is useful when a single logical resource needs to be accessed by two tasks with disjoint rights: one task holds the write capability, the other holds the read capability, and neither can exercise the other's rights.

7. Capability Revocation

Revocation is the ability to permanently invalidate a capability so that no future operations using its token succeed. LateralusOS provides two revocation primitives: cap::revoke(cap), which invalidates a specific capability by token, and cap::revoke_tree(cap), which invalidates the capability and all of its descendants in the delegation tree.

```
// Revocation example: temporary file access for a sandboxed parser
fn parse_untrusted(fs: Cap<FsRoot>, path: &str, data: &[u8]) -> ParseResult {
    let tmp_cap = fs |> cap::open(path, RIGHT_READ);
    let parser_cap = tmp_cap |> cap::restrict(path, RIGHT_READ);
    let result = sandboxed_parser(parser_cap); // parser gets the cap
    // cap has been moved into sandboxed_parser;
    // revoke the entire delegation subtree:
    cap::revoke_tree_by_path(path); // kernel-side revocation
    result
}
```

The kernel implements revocation by marking the capability table entry as invalid. Because the kernel validates the table entry on every syscall, any attempt to use a revoked token returns Err(CapError::Revoked) immediately. No scanning of process address spaces is required; the kernel's per-syscall check is sufficient.

Tree revocation is O(d) in the depth of the delegation tree, not O(n) in the number of processes. This is because the kernel maintains a parent-pointer linked list in each capability table entry. Revoking a node recursively follows child pointers; because the tree is bounded by the depth of delegation chains (typically 3-5 in LateralusOS workloads), tree revocation is fast in practice.

A notable edge case: when a task exits, the kernel automatically revokes all capabilities in its handle table, including any that were derived from root capabilities. This prevents zombie capabilities that outlive their owning task. The Lateralus runtime calls the EXIT syscall only after the linear type drop handlers have run, so all Cap<T> values are either properly closed or revoked before the process table entry is freed.

8. Capability-Passing Through Pipelines

The Lateralus |> operator interacts cleanly with Cap<T> because both are defined in terms of move semantics. A pipeline stage that requires a capability declares it as a parameter; when the caller pipes a capability into the stage, the move is explicit in the type signature. The compiler validates that the capability is not used again after being piped.

```
// A pipeline of capability-accepting stages
```

```
fn audit_log_pipeline(fs: Cap<FsRoot>) -> Result<(), AuditError> {
    fs
    |> cap::open("/var/log/audit.log", RIGHT_READ | RIGHT_SEEK)
    |?> fs::seek_to_end
    |?> parse_audit_records
    |?> flag_anomalies
    |?> write_report("/var/report/anomalies.json")
}
```

The |?> operator is the error-propagating variant. When a stage returns Err, downstream stages are skipped and the capability is dropped automatically, triggering the cap_close syscall. This ensures that error paths do not leak capabilities; the clean path and the error path both correctly release resources.

The cap_pipe() primitive extends this model for cases where multiple capabilities are needed across stages. It accepts a tuple of capabilities and threads them through a stage sequence, splitting or combining as needed.

```
// cap_pipe with multiple capabilities
fn copy_file(
    src_cap: Cap<FileResource>,
    dst_cap: Cap<FileResource>,
) -> Result<u64, IoError> {
    cap_pipe!(
        (src_cap, dst_cap)
        |?> |(s, d)| fs::copy_chunks(s, d, CHUNK_SIZE)
        |?> |(_, _)| Ok(CHUNK_SIZE as u64)
    )
}
```

This design ensures that the least-privilege principle is not just a policy stated in documentation but a property enforced by the type system. A function that accepts Cap<FileResource> with RIGHT_READ cannot write to the file regardless of what the function body attempts, because any write syscall will fail the kernel rights check at the ecall boundary.

9. Filesystem Capabilities

Filesystem capabilities in LateralusOS cover five categories of operations: data access (read, write, truncate, seek), metadata access (stat, lstat), directory operations (readdir, mkdir, rmdir), linking (link, unlink, rename), and extended attributes (getxattr, setxattr). Each is gated by a distinct right bit, allowing fine-grained policies.

```
// Example: read-only, stat-only cap for a config watcher
fn make_config_watcher_cap(fs: Cap<FsRoot>) -> Cap<FileResource> {
    fs |> cap::open("/etc/app/", RIGHT_READ | RIGHT_STAT)
       |> expect("failed to create config watcher cap")
}

// The watcher cannot write or unlink; those bits are absent.
fn watch_config(cap: Cap<FileResource>) {
    loop {
        let mtime = cap |> fs::stat |> |s| s.mtime;
        if mtime != last_mtime { reload_config(cap); }
        sleep_ms(500);
    }
}
```

Directories are first-class objects in the capability model. A capability for a directory path implicitly covers all files under that path, but only if the directory capability includes the RIGHT_READ flag.

Accessing a file under a directory without a covering capability returns Err(CapError::NotCovered). This prevents path traversal attacks: a component that holds Cap(/tmp/sandbox) cannot access /tmp/sandbox/../etc/passwd because the kernel resolves the path and verifies coverage after normalization.

The cap::watch() API wraps the filesystem capability and returns a FsWatcher<T> that receives inotify-like events when files under the covered path change. The watcher holds a reference to the underlying capability token, so it is automatically invalidated when the capability is revoked. This is a significant advantage over POSIX inotify, where watch descriptors outlive the permission to access the file.

Hard links and symbolic links are handled carefully. Creating a hard link requires a capability covering the destination directory with both RIGHT_WRITE and a new RIGHT_LINK bit. Symbolic links are followed transparently by the kernel capability check: when a component presents a capability covering a symlink target's resolved path, access is granted. Symlinks that escape the capability's covered subtree are rejected at the kernel level, preventing symlink-based path traversal.

10. Network Capabilities

Network capabilities gate socket creation, binding, connection, and data transfer. The capability model extends to network namespaces: a Cap<NetRoot> covers only the network namespace it was issued in, preventing cross-namespace socket operations even if a process holds namespace-crossing credentials.

```
// Full example: HTTP server with minimal network capability
fn run_http_server(net: Cap<NetRoot>) -> Result<(), ServerError> {
    let listener_cap = net
        |> cap::bind_tcp("127.0.0.1:8080",
                         RIGHT_ACCEPT | RIGHT_RECV | RIGHT_SEND)
        |?> |c| Ok(c);

    loop {
        let conn_cap = listener_cap |?> net::accept;  // Cap<TcpStream>
        spawn(|| handle_conn(conn_cap));  // cap moved to child task
    }
}
```

Each accepted connection yields a separate Cap<TcpStream>. The connection capability carries the peer address baked into its scope field, so a compromised handler task cannot send data to an arbitrary peer by presenting its connection capability to a different socket syscall. The peer address is verified on every send operation.

UDP sockets use a similar model. A Cap<UdpSocket> is created with a destination address and port range restriction. Sending to addresses outside the range returns Err(CapError::AddressNotCovered). This allows a DNS resolver task to hold a capability that covers only port 53 on the configured upstream servers, with no ability to send to arbitrary hosts.

TLS termination interacts with capabilities at the record-layer level. LateralusOS provides a cap::tls_wrap(stream_cap, tls_config) API that returns a Cap<TlsStream>. The TLS capability inherits the rights of the underlying stream capability; reading or writing the TLS stream requires the same RIGHT_RECV and RIGHT_SEND bits as the underlying TCP stream, providing consistent rights semantics across protocol layers.

11. Hardware Capabilities

LateralusOS permits userspace device drivers by granting hardware capabilities to trusted processes. Three hardware capability types are currently defined: Cap<DmaRegion> for direct memory access, Cap<IrqLine> for interrupt delivery, and Cap<IoPort> for x86-style port-mapped I/O (retained for compatibility; RV64 typically uses MMIO).

```
// Userspace NVMe driver initialization
fn init_nvme_driver(
    hw: Cap<HwRoot>,
    pci_bdf: PciBdf,
) -> Result<NvmeDriver, HwError> {
    let bar0 = hw |> cap::mmio_map(pci_bdf, bar=0,
                                   size=0x4000,
                                   rights=RIGHT_READ | RIGHT_WRITE);
    let irq  = hw |> cap::irq_bind(pci_bdf.irq_line(), RIGHT_IRQ_MASK);
    let dma  = hw |> cap::dma_alloc(pages=64, RIGHT_DMA_MAP);
    NvmeDriver::new(bar0, irq, dma)
}
```

DMA capabilities cover a specific physical memory region and include IOMMU programming by the kernel. When a Cap<DmaRegion> is created, the kernel programs the IOMMU to allow DMA only to the covered physical pages. If the driver is compromised and attempts to configure the device to DMA to kernel memory, the IOMMU raises a fault and the kernel kills the driver task and revokes the hardware capability.

Interrupt capabilities allow a userspace driver to wait for and acknowledge hardware interrupts without full kernel involvement. The irq::wait(irq_cap) call blocks the calling task until the interrupt fires; the kernel delivers the interrupt event by unblocking the task and returning the interrupt vector number. Acknowledging the interrupt requires calling irq::ack(irq_cap), which is gated on RIGHT_IRQ_MASK. A task that holds only the wait right cannot acknowledge interrupts, which prevents a compromised driver from masking interrupts and causing a denial-of-service condition.

MMIO memory-mapped capability regions are backed by PMP (physical memory protection) entries on RV64 or EPT pages on x86. The kernel configures the hardware protection unit to allow the holding task access to only the mapped physical range. Writes outside the capability's MMIO range trap to the kernel and return an access fault to the driver task, providing hardware-enforced MMIO isolation between device drivers.

12. The cap_pipe() Primitive

Most pipelines in Lateralus involve a single value flowing through stages. When security contexts require multiple capabilities across a pipeline, the cap_pipe!() macro provides a syntax for threading multiple capabilities in parallel.

```
// cap_pipe macro expansion (simplified)
macro_rules! cap_pipe {
    (($($cap:expr),+) |?> $stage:expr) => {{
        match ($($cap),+) {
            caps => $stage(caps)
        }
    }};
    (($($cap:expr),+) |?> $stage:expr $( |?> $rest:expr )*) => {{
        match $stage(($($cap),+)) {
            Ok(next_caps) => cap_pipe!(next_caps $( |?> $rest )*),
            Err(e) => Err(e),
```

```
        }
    }};
}
```

A concrete use case is a database connection setup that requires both a network capability (to connect to the database server) and a filesystem capability (to read the TLS client certificate). The two capabilities are bundled into a tuple and piped through the setup stages.

```
fn db_connect(
    net: Cap<NetRoot>,
    fs:  Cap<FsRoot>,
) -> Result<Cap<DbConnection>, DbError> {
    let cert_cap = fs |> cap::open("/etc/certs/client.pem", RIGHT_READ);
    let net_cap  = net |> cap::connect_tcp("db.internal:5432",
                                           RIGHT_SEND | RIGHT_RECV);
    cap_pipe!(
        (cert_cap, net_cap)
        |?> |(cert, stream)| tls::connect(stream, cert)
        |?> |tls_stream| db::handshake(tls_stream)
    )
}
```

The macro is syntactic sugar; the expansion is a sequence of nested match expressions. The compiler reduces the expansion at compile time, and the resulting code is equivalent to a manually written chain of function calls. The macro exists purely to reduce boilerplate and make the pipeline structure visible to human readers.

13. LateralusOS Kernel Enforcement

The kernel capability enforcement path is intentionally minimal: it consists of a hash-table lookup by token high 64 bits, a rights-bit AND check, and a scope-path prefix check for filesystem capabilities. The hash table uses open addressing with a load factor of 0.5; at this load factor, the expected number of probes is 1.5, giving a nearly constant-time lookup regardless of the number of active capabilities.

```
// Kernel cap_check pseudocode (C, kernel context)
int cap_check(u128 token, u64 required_rights, const char *path) {
    u64 obj_id = token >> 64;
    u64 tok_hi = token & 0xFFFFFFFFFFFFFFFFULL;
    CapEntry *e = cap_table_lookup(obj_id);
    if (!e || e->token_hi != tok_hi) return -CAP_ERR_INVALID;
    if (e->revoked)                  return -CAP_ERR_REVOKED;
    if ((e->rights & required_rights) != required_rights)
                                     return -CAP_ERR_DENIED;
    if (path && !path_covered(e->scope, path))
                                     return -CAP_ERR_NOT_COVERED;
    return 0;
}
```

The path_covered() function performs a normalized path prefix check after resolving all symlinks using the kernel's VFS layer. Normalization eliminates .. components and multiple slashes before the prefix check, closing the symlink-based traversal attack. The VFS resolution result is cached in the dentry cache; for hot paths, the scope check adds only a dentry cache lookup to the critical path.

The kernel capability table is stored in a dedicated memory region protected by PMP. Userspace reads from this region trap to the kernel; writes are not mapped at all. The table itself is not exported via any /proc or /sys interface; the only way to inspect a capability is via the cap_inspect syscall, which requires presenting a valid capability token with RIGHT_INSPECT. This prevents enumeration attacks where an

attacker tries to discover valid tokens by scanning the capability namespace.

Each capability table entry is 64 bytes: 16 bytes for the token, 8 bytes for the rights field, 8 bytes for the reference count, 8 bytes for the parent pointer (for tree revocation), 4 bytes for flags including the revoked bit, and 20 bytes for the scope path hash (a SHA-1 of the normalized scope string, sufficient for collision resistance in the kernel context).

14. Comparison with DAC and MAC

Discretionary Access Control (DAC), the model used by POSIX systems, grants permissions based on the owner and group of each file and the UID/GID of the requesting process. Permissions are stored with the resource (in the inode mode bits) and checked at each access. The key weakness of DAC is ambient authority: a process inherits the full permissions of its UID, and any code running in that process can exercise those permissions, regardless of whether the specific code path was intended to do so.

Mandatory Access Control (MAC), as implemented by SELinux and AppArmor, adds a second layer of policy that restricts what operations a process can perform based on its security label. MAC policy is centrally defined by an administrator and cannot be modified by the process itself. MAC is effective at enforcing system-wide policies but does not solve the confused deputy problem within a process: a setuid process with a given SELinux label can be tricked into performing any operation that label permits.

- DAC: identity-based, ambient authority, confused deputy possible
- MAC: label-based, centrally administered, confused deputy within label possible
- Capabilities: token-based, explicit authority, confused deputy structurally impossible

The key advantage of capabilities over both DAC and MAC is that authority must be explicitly passed to each code path that uses it. There is no ambient authority that can be accidentally exercised. A library function that opens a file must receive a filesystem capability from its caller; it cannot open an arbitrary file using the process's ambient UID.

The key disadvantage is operational complexity: every permission must be explicitly threaded through the call graph. LateralusOS mitigates this via the Cap<T> type and the pipeline operator, which make capability threading syntactically lightweight. The startup-time root capability pattern also reduces the ceremony needed for simple applications that do not need fine-grained least-privilege.

15. The Confused Deputy Problem

Hardy's 1988 confused deputy paper described a compiler service that ran with elevated privileges because it needed to write to the billing file. An attacker could trick the compiler into overwriting the billing file with a specially crafted source file, because the compiler exercised its billing-file authority whenever it decided to write output, without distinguishing between "writing because the user asked me to" and "writing to the billing file because that is a special internal operation."

Under the capability model, the compiler would hold two separate capabilities: one for the user's output directory (narrowed from the user's filesystem capability) and one for the billing file (narrowed from the system administrator's capability). The compiler cannot write to the billing file using the user's output capability because the kernel scope check would reject it. The compiler cannot write to the user's output directory using the billing capability because the scope does not cover the output path.

```
// Confused deputy: impossible in LateralusOS
fn compile(
```

```
    user_out_cap: Cap<FileResource>,  // scope: /home/user/out/, RIGHT_WRITE
    billing_cap:  Cap<FileResource>,  // scope: /var/billing, RIGHT_WRITE
    source_file:  &[u8],
) -> Result<(), CompileError> {
    // Attacker cannot cause user_out_cap to access /var/billing:
    // kernel scope check rejects any path not under /home/user/out/
    let obj = compile_to_object(source_file);
    user_out_cap |?> fs::write("output.o", &obj);
    billing_cap  |?> fs::append("compile.log", &billing_entry(source_file))
}
```

The structural impossibility of the confused deputy problem under capabilities is a theorem, not a policy. It does not require the programmer to be vigilant about which capability they use; the compiler rejects any attempt to use the wrong capability for a path not covered by that capability's scope. Security properties that are theorems are more robust than properties that are policies.

16. POSIX Comparison

POSIX provides a file-descriptor abstraction that has superficial similarity to capabilities: a file descriptor is an opaque integer that grants access to a file, and it can be passed to child processes via inheritance or to other processes via SCM_RIGHTS socket messages. However, POSIX file descriptors are not unforgeable (integers can be guessed and reused after close), are not linear (they can be duplicated with dup()), and do not carry fine-grained rights (only the open flags from open() are recorded, not a bitfield of specific operations).

Capsicum improves on POSIX by adding a rights bitmask to file descriptors and entering a capability mode that disables global namespace operations. Capsicum capabilities are not linear and can be duplicated via dup(); a duplicated Capsicum capability has the same rights as the original. LateralusOS's linear type enforcement prevents duplication entirely at the language level.

- POSIX fd: mutable integer, dupable, no rights bitfield, ambient UID
- Capsicum: rights bitfield on fd, no linearity, no delegation tree
- Fuchsia handle: rights bitfield, transfer via channel, no linearity
- LateralusOS Cap: rights bitfield, linear type, delegation tree, compile-time enforcement

The SCM_RIGHTS mechanism in POSIX sends file descriptors over Unix domain sockets. This provides a capability-passing mechanism but requires manual discipline: the sender must close its copy of the descriptor after sending to avoid retaining access. LateralusOS's linear type system makes this discipline automatic: the pipeline operator moves the capability token, and the compiler rejects any attempt to use it after the move.

17. Benchmark Results

We measured the overhead of capability checking on an RV64GC core (SiFive U74 at 1 GHz) running LateralusOS 0.12. The benchmark measures the round-trip latency of a filesystem read syscall with and without capability enforcement. All measurements are the median of 10,000 iterations, with the standard deviation reported.

```
// Benchmark results (nanoseconds, RV64GC @ 1 GHz)
Operation                      No-cap (ns)   With-cap (ns)   Overhead
-------------------------------------------------------------------
fs::read (hot dentry cache)        142            180            38 ns
fs::write (hot page cache)         168            207            39 ns
```

```
net::recv (loopback)              210           251           41 ns
net::send (loopback)              198           239           41 ns
cap::open (cold)                  --            890          N/A
cap::restrict                     --            210          N/A
cap::delegate                     --           1240          N/A
cap::revoke                       --            420          N/A
cap::revoke_tree (depth 4)        --           1680          N/A
```

The 38-41 ns per-syscall overhead consists of two hash-table probes (token lookup and rights check) and a dentry-cache prefix comparison. The dentry cache hit rate for the benchmark workload was 99.7%, so the normalized path resolution cost is negligible. On a cold dentry cache, the overhead rises to approximately 180 ns, dominated by VFS path resolution.

Capability delegation is the most expensive operation at 1240 ns because it requires updating two process handle tables (source and destination) under a spinlock. Revocation of a single capability costs 420 ns; tree revocation of a depth-4 tree costs 1680 ns. These operations are expected to be infrequent in production workloads and are not on the critical path of any I/O-intensive application.

Compared to a Linux system with SELinux enforcing mode, LateralusOS's capability check adds 38 ns versus SELinux's 55 ns per syscall (measured on equivalent hardware). The difference arises because SELinux's policy engine evaluates a rule list while LateralusOS's check is a fixed 3-step hash-table lookup. LateralusOS does not yet implement an SELinux-equivalent audit log, which would add approximately 15 ns to the per-syscall cost.

18. Attack Surface Analysis

The trusted computing base (TCB) of the LateralusOS capability system consists of four components: the kernel capability table implementation (1,400 LOC), the capability syscall handlers (800 LOC), the PMP/IOMMU programming code (600 LOC), and the Lateralus compiler's linearity checker (2,200 LOC). These are the only components whose bugs can compromise the security guarantees.

The most dangerous class of bugs in the kernel capability code is a use-after-free on the capability table entry. If the entry is freed before the syscall handler finishes reading the rights field, a malicious task could reclaim the entry with different rights and win a race. LateralusOS addresses this with RCU (read-copy-update) semantics on the capability table: table entries are freed only after a grace period during which no active syscall handler can hold a reference.

Integer overflow in the rights bitfield AND check is a second class of concern. The check (entry.rights & required_rights) != required_rights is simple but must be performed on unsigned integers to avoid undefined behavior in C. LateralusOS's kernel is written in Lateralus itself for the userspace-adjacent layers, providing overflow-by-default protection for arithmetic on rights fields.

The compiler's linearity checker is part of the TCB because a bug there could allow a capability to be used after delegation, giving a task access to a resource it should no longer hold. The linearity checker is currently 550 lines of Lateralus and is being prepared for formal verification using a proof assistant (see Section 19).

19. Limitations and Future Work

The current LateralusOS capability model has several known limitations. First, the scope field of a filesystem capability is a simple path prefix, not a general predicate. It is not possible to express "access to all .log files in /var/" without granting access to the entire /var/ subtree. Future work will explore a pattern-based scope language that allows glob expressions as capability scopes.

Second, network capabilities do not currently cover IPv6 multicast and broadcast addresses. A capability covering 0.0.0.0/0 can be used to send to multicast groups, which may violate the principle of least privilege for applications that only need unicast communication. A future version will add separate right bits for multicast and broadcast.

Third, the delegation tree is maintained in-memory only. If the system crashes and restarts, delegation relationships are lost. Capabilities issued before the crash are invalid after restart because the kernel generates a new random seed for token high bits at boot. This prevents stale token replay attacks but means that applications must re-acquire capabilities at each boot. Persistent capabilities backed by a TPM-sealed store are planned for a future release.

Formal verification of the kernel capability table code using RISC-V assembly verification tools (rv-star, CompCert) is underway. The goal is a machine-checked proof that the cap_check function correctly implements the monotonic narrowing invariant and that no code path allows rights escalation. Initial results from automated property testing with a randomized syscall fuzzer have found zero rights-escalation bugs in 10 million test cases.

20. Conclusion

We have presented the LateralusOS capability-based security model, centered on the Cap<T> linear type. The model prevents the confused deputy problem structurally, enforces least-privilege policies at compile time, and adds a median overhead of 38 ns per syscall on RV64GC hardware. The integration with Lateralus's pipeline operator makes capability passing syntactically lightweight, reducing the programmer burden compared to manual capability threading in C.

The system draws on a rich lineage of capability research from Dennis and Van Horn (1966) through EROS, seL4, Capsicum, and Fuchsia, and extends that tradition with compile-time linearity enforcement that previous systems achieved only at runtime or not at all. We believe that systems-programming languages are the right place to enforce security invariants because the compiler's type system can provide guarantees that are impossible to achieve with runtime-only enforcement.

LateralusOS and the capability implementation are open source. The kernel capability code, the Lateralus standard library Cap<T> implementation, and the linearity checker are all available at the project repository. We welcome contributions to the formal verification effort and to the network capability extensions described in Section 19.