Practical Principle of Least Privilege for Secure Embedded Systems

Samuel Jero* Juliana Furgala* Runyu Pan† Phani Kishore Gadepalli† Alexandra Clifford‡§
Bite Ye† Roger Khazan* Bryan C. Ward* Gabriel Parmer† Richard Skowyra*
*MIT Lincoln Laboratory, †The George Washington University, ‡Draper Laboratory

***Abstract*—Many embedded systems have evolved from simple bare-metal control systems to highly complex network-connected systems. These systems increasingly demand rich and feature-full operating-systems (OS) functionalities. Furthermore, the network connectedness offers attack vectors that require stronger security designs. To that end, this paper defines a prototypical RTOS API called Patina that provides services common in feature-rich OSes (*e.g.,* Linux) but absent in more trustworthy μ-kernel-based systems. Examples of such services include communication channels, timers, event management, and synchronization. Two Patina implementations are presented, one on Composite and the other on seL4, each of which is designed based on the Principle of Least Privilege (PoLP) to increase system security. This paper describes how each of these μ-kernels affect the PoLP-based design, as well as discusses security and performance tradeoffs in the two implementations. Results of comprehensive evaluations demonstrate that the performance of the PoLP-based implementation of Patina offers comparable or superior performance to Linux, while offering heightened isolation.**

I. Introduction

Embedded systems must manage the competing forces of *increasing workload complexity* such as autonomous driving, and the need for *strong security* due to the criticality of their functionality. To enable their rich functionality, such systems are increasingly network connected (*e.g.,* (I)IoT), must handle diverse input sources (*e.g.,* cameras, lidar, sensors), and must carry out nuanced control processing tasks. Further, many services are increasingly being consolidated on a common computing platform. While such systems offer the promise of new technologies and features, their network exposure and advanced software capabilities pose dangerous new attack vectors for cyber criminals. Towards that end, it is imperative that future embedded systems are built upon secure and trustworthy OSes that can support demanding real-time workloads.

These workloads are increasingly being migrated from bare-metal or embedded RTOS systems to more feature-full operating systems such as Linux. For example, SpaceX famously controls many of their systems, such as the Falcon rocket and Dragon capsule, with Linux with the `PREEMPT_RT` patch [1]. However, large complex monolithic operating systems are also subject to more vulnerabilities – borne out by their constant stream of CVEs [2] – given their massive size and complexity. For systems that control the physical environment, such compromises can lead to damage or human harm.

An important security design goal is the Principle of Least Privilege (PoLP) in which "Every program and every user of the system should operate using the least set of privileges necessary to complete the job" [3]. A consequence of this principle is that the scope of any compromise is restricted only to the small set of accessible resources from the compromised functionality. It is often paired with a focus on software simplicity (economy of mechanism) to provide software that is more easily certified and resilient to attack. Toward these goals, μ-kernel-based OSes have limited functionality, and implement higher-level features as isolated, user-level services. As such, μ-kernels enable highly trustworthy designs, and seL4 demonstrates this as the first formally verified OS [4].

However, given the minimalist μ-kernel architecture, common services must be implemented in userspace. In practice, it is common for μ-kernels to be employed as a separation kernel, with most applications executing in virtual machines. Real-time and embedded applications often avoid complex APIs such as POSIX, but require a basic interface including threads, message passing, timer-based activation, and synchronization. These benefit from a trustworthy, simple interface and implementation, rather than running in a complex VM.

This paper investigates RTOS abstraction layers on top of μ-kernels that are designed to enforce PoLP *within* the abstraction layer to strengthen system security. We call our RTOS API Patina[1], and it is designed to be a prototypical RTOS API. We designed a new small and simple μ-kernel-agnostic API that could be efficiently implemented on a variety of μ-kernels. In comparison, prior OS APIs are either (i) incredibly low level, forcing developers to deal with undue complexity (*e.g.,* raw SeL4 or Composite), (ii) bloated and complex (*e.g.,* POSIX), or (iii) are designed for a single shared address space (*e.g.,* FreeRTOS). In the remainder of this paper, we refer to an implementation of this API as a Patina.

A naïve approach to RTOS-API implementation on modern μ-kernels is to place all RTOS services in a single protection domain and use IPC-based service invocations. However, in this design a fault or compromise in any service (*e.g.,* communication) could impact all services and/or applications. Instead, in this paper we focus on RTOS-API implementations

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

We'd also like to thank NSF for support through CNS-1815690 and CPS-1837382. The views of this paper do not necessarily reflect the NSF.

§Work done while at MIT Lincoln Laboratory.

[1]A patina is a thin layer on top of a surface that is often protective.

(specifically Patina implementations) that separate system functionality into separate, isolated user-level services, while focusing on simplicity of implementation. As such, our focus is enabling more fine-grained granularity of resource access to RTOS services, thus constraining the impact of any failures. The core scientific question is if a PoLP-optimized RTOS can provide real-time, predictable performance, and common-case performance competitive with existing systems. Significant results have laid the groundwork for a PoLP RTOS: (1) Mehnert [5] showed that user-level, isolated system services can provide response times on the order of *kernel-resident* logic, and (2) Slite [6] demonstrates that user-level scheduling can have similar or better performance to kernel-resident scheduling. This paper seeks to answer if an RTOS consisting of many higher-level system services can maintain strong predictability, while also achieving sufficient average performance. Note that such a PoLP-driven RTOS can co-exist with virtual machines to provide legacy execution environments.

To demonstrate both the feasibility of developing PoLP-API implementations, as well as to compare and contrast design methodologies, we implemented PoLP-focused Patinas on two different μ-kernels, Composite and seL4. These different μ-kernels have differing design philosophies and mechanisms, which has yielded different Patina designs. Based upon these two independently developed Patinas, we discuss (i) design commonalities, (ii) performance and security considerations of different design decisions and (iii) lessons learned developing PoLP-focused services on differing μ-kernel architectures.

After describing relevant security challenges (§ II) and background (§ III), this paper makes the following contributions.

- We describe two independently developed Patinas on two different μ-kernels, seL4 and Composite. (§ IV)
- We discuss the design and implementation of Patina on each, guided by the principal of least privilege. (§ IV)
- We evaluate both Patinas with respect to their predictability and performance, and find that they provide significant quantitative benefits in addition to stronger isolation. (§ V)
- We discuss different Patina design decisions in each implementation based on security/performance tradeoffs, as well as the underlying μ-kernel design. (§ VI)

II. Security Challenges

The PoLP is an important security-design principle. However, it does not protect against all threats. It is therefore important to consider the attack vectors and threat models that the PoLP is designed to address within our OS Patina.

When applying the PoLP within an OS, the privileges and capabilities of processes and system services are reduced. This limits what an attacker can do if they are able to compromise a system process. For example, a compromise of a vehicle infotainment system should not enable the attacker to hijack control of the steering of the vehicle.

We therefore consider a threat model in which there is a potentially malicious user-space process. Such a malicious process may seek to exfiltrate data, corrupt system integrity, achieve adversarial remote control, *etc.* To some degree, a threat model based on malicious applications may appear to be admitting defeat at the outset. Clearly, a compromised application can adversely impact the system by refusing to perform its role, *e.g.,* by ignoring service requests or consuming resources (*e.g.,* CPU cycles) without generating useful output. However, stopping possible compromise across all system services is an effectively impossible endeavor. Even formal verification relies on assumptions that have been shown able to be violated in the real-world. For example, the Spectre [7] and Rowhammer [8] attacks demonstrated the danger of assuming that hardware behaves according to its specification. In addition, attacks based on impersonating legitimate operators (*e.g.,* the credential theft [9]) will not be stopped by bug-free code, since attackers are operating outside the verification boundary.

This threat model of assuming that a process is potentially malicious is also supported by a number of different attack vectors. While there are myriad ways in which an attacker could compromise a process on the system, they fall into two broad categories: (i) malicious input, which exploits vulnerabilities in code, often to hijack control of the victim process; and (ii) malicious code, such as software installed on the system that was developed by an untrusted party or subject to a software supply-chain threat.[2] These attack classes demonstrate the relevance of this threat model, though the specific mechanism employed by an attacker is inconsequential to our model. While there are defenses that seek to mitigate some of these attack techniques, they are not perfect, and attackers constantly evolve to bypass new and stronger defenses. Such defenses are complementary to PoLP-based mechanisms.

Additionally, we consider the μ-kernel itself to be benign and not subject to compromise. μ-kernels are minimal, highly trusted, and in the case of seL4, formally verified [4]. In our OS Patinas, there are user-mode services providing system functionality not implemented by the μ-kernel itself. These services are considered to be potentially buggy, and therefore subject to compromise, but benign.

With this threat model and motivation, there are several key security and performance challenges associated with developing a PoLP-optimized embedded system:

- **Predictability.** Least-privilege enforcement must not lead to temporal violations of real-time requirements. Any least-privilege policy must result in predictable execution regardless of the complexity or nuance of that policy.
- **Minimality.** Embedded systems are resource-constrained and lack a substantial margin for the addition of new capabilities. Least-privilege enforcement must minimize its impact on legitimate system operations, which often constitute the vast majority of normal execution.
- **Granularity.** Privilege, especially with respect to resource access and shared data structures (*e.g.,* for synchronization) must be as fine-grained as possible in order to ensure that any violation of intended application semantics will be detected and prohibited. However, finer granularities of enforcement may also make enforcement more intrusive

[2]Hardware-based threats, such as Rowhammer [8], may also enable process exploitation, but we consider such threats outside the scope of this work.

(reducing minimality) or require complex checks and meta-data operations (reducing predictability).

- **Recovery.** Small components offer the potential to recover faulty or malicious processes efficiently [10]. However, the system must be designed to enable such recovery without otherwise compromising the security of the system.

III. μ-kernel Background & Related Work

Capability-based models are a strong fit for enforcing PoLP as they support fine-grained and efficient access control for kernel resources. Figure 1 depicts different OS structures among a self-contained RTOS with monolithic support for core services (a), and two instances of modern μ-kernels (b, c) that decentralize services while applying the PoLP. Note the focus in (b) and (c) not only on inter-application isolation, but also on the PoLP-focused isolation between system services.

A. Background and μ-Kernel Foundations

Capability-based kernels. Modern μ-kernels, inspired by Eros [11], use *capability-based addressing to access all kernel resources* [12]. A *capability* is an unforgeable token denoting a principal's access to a resource. They enable *delegating* access to the resource by copying the capability to another principal, and *revoking* capabilities previously delegated, thus removing access to the corresponding kernel resource.

We describe principals that have capabilities as *protection domains* that include a virtual address space (provided by page tables) and the principal's capability tables. Threads executing in a protection domain are confined to the resource accesses allowed by the protection domain. Inter-Process Communication (IPC) provides inter-protection-domain coordination, enabling client requests to be handled by logic in a different protection domain than the client, thus with different privileges. Capability-based access control enables a composable, efficient means of manipulating the access to kernel resources and the construction of protection domains. IPC enables the separation of concerns since protection domains specialize to provide mediated access to services.

Kernel minimality. The guiding principle for μ-kernels [13] is *minimality*. "A concept is tolerated inside the μ-kernel only if moving it outside the kernel [...] would prevent the implementation of the system's required functionality." μ-kernels implement device drivers in user level, which requires translating interrupts into kernel-instigated IPC. System policies for networking, memory management, and time management are implemented in user-level protection domains. Of note, *scheduling policy and blocking semantics* have traditionally been bound to the kernel, even in L4 variants.

Exporting policies to user level has a number of benefits. It enables configurability of core system policies, minimizes the size of the kernel that *must be trusted by all system execution*, and enables the separation of concerns for different policies, each implemented in different protection domains. In doing so, it encourages the application of the PoLP.

The removal of memory management (esp. allocation) from the kernel is of particular note. The user-level management of kernel memory [14] is *safely* enabled with kernel-provided memory *retyping* facilities. Memory typed as *frames* are otherwise unusable, but are tracked using capabilities. They can be retyped into various forms of kernel memory or retyped into user-accessible virtual memory. This is safe as protection domains can use only capability-accessible memory, and memory can only have a single type (thus protecting kernel data structures from user-level access).

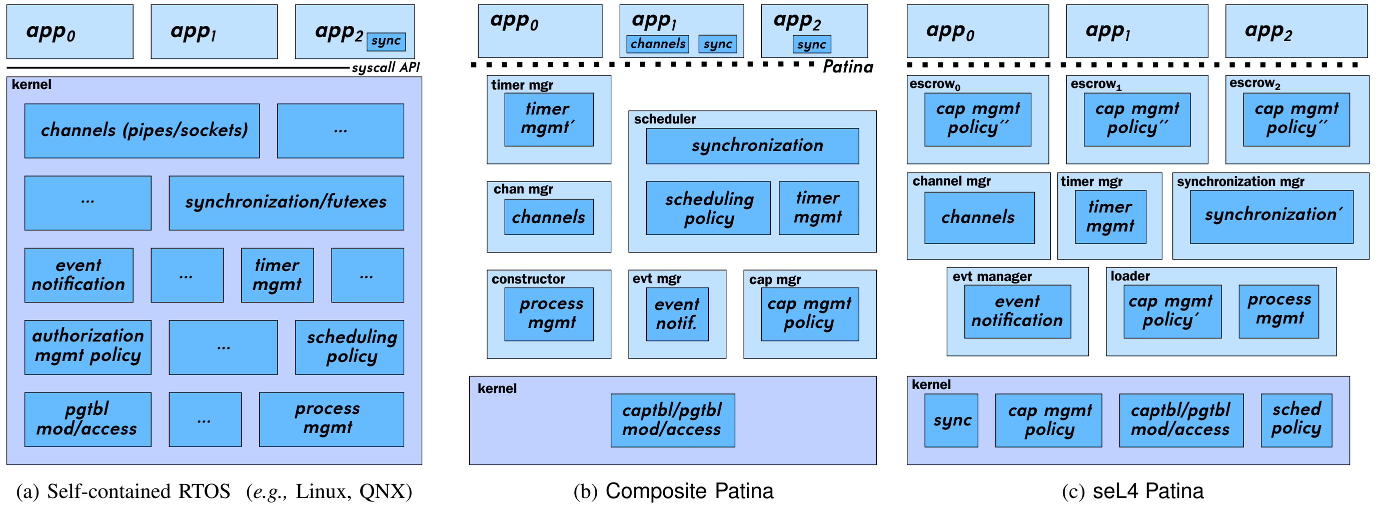
Benefits of isolation. The errant effects of a faulty or compromised service in monolithic RTOSes (Figure 1(a)) can impact all applications. In contrast, both Patina implementations move system services into separate protection domains. A failure in one is constrained to its logic, data structures, and any service requests. For example, a failure in the channel service, being used to supply navigational commands from a radio to high-level drone software, will not directly impact a critical safety-of-flight device driver that keeps the drone flying. Though beyond the scope of this research, such recovery or reset mechanisms (*e.g.,* using exception models [15], interface-driven recovery [10], or redundant execution [16]) are fundamentally dependent on isolation.

B. Related work

Predictable μ-kernel implementation. μ-kernels are a natural choice for real-time and embedded systems as the increased isolation they provide is an asset for high-confidence computation. Previous work [5] demonstrates that the latency of translating interrupts into user level IPC is not prohibitive while [17] demonstrates that interrupt-scheduling policy can also be provided at user level. Composite has demonstrated the ability to scale predictability guarantees up to multi-core systems [18]. Blackham [19] demonstrates the automated WCET calculation of a μ-kernel. Unfortunately, prior research has not demonstrated that the end-to-end predictability of a multi-protection-domain RTOS is possible and reasonable.

Component-based Environments. The SawMill multi-server OS [20] built on L4, and the FLUX OSKit [21] decompose existing monolithic systems into their constituent parts, and execute them in (potentially) isolated protection domains. Similarly, Composite is a component-based OS capable of supporting webserver functionality in around 25 components [22], though without the focus on RTOS functionality or predictability. Camkes [23] is a component-based development environment for seL4 used to construct the initial set of protection domains, kernel resources, and connections among them. OS personalities in Workplace OS [24], GrailOS [25], and Exokernel [26] are custom implementations of existing OS abstractions (*e.g.,* POSIX, Win32). In this work, we define a new OS abstraction, Patina, and implement two Patina personalities on different μ-kernels. Our focus is on the *design* of these implementations and how applying PoLP design principles provides isolation and impacts performance.

L4Re [27] and Nova [28] are runtime environments with a strong focus on providing facilities for multiplexing I/O and memory across virtual machines while providing per-VM



(a) Self-contained RTOS (*e.g.,* Linux, QNX) (b) Composite Patina (c) seL4 Patina

Fig. 1: Differing system structures for an RTOS API. The dotted line is the Patina API, light-blue boxes are protection domains, dark blue are functional modules, and the purple box is the kernel.

VMMs. This focus on VM support is complementary to the PoLP-focused, high-confidence RTOS of this paper.

Patina, to the best of our knowledge, is the first system design to thoroughly apply the PoLP to an RTOS implementation. Further, we've demonstrated in two different systems, that more aggressively pursuing the PoLP for an RTOS does *not* prohibitively impact performance nor real-time predictability.

IV. Patina Design and Implementation

We have developed two PoLP-based implementations of Patina: one on Composite and one on seL4 For Composite, this is reflected in the distribution of responsibility and authority for the management of different system abstractions among isolated *components*. For seL4, this manifests in the development of self-contained processes to isolate and protect a process's memory and capabilities from all other parts of the system. First, we introduce our Composite and seL4 Patinas and provide an overview of the Patina API.

A. Architecture Overview

Both the Composite kernel and the seL4 kernel expose difficult to use, very low-level system call APIs. For example, creating a new thread involves retyping memory to be used for the kernel's thread object, retyping and using (potentially multiple levels of) capability-table nodes to provide the capability table for the thread, retyping and using multiple levels of page-table nodes to provide virtual memory for the thread, and, finally, the association of the thread with a protection domain. This complexity is particularly pronounced for Composite since the kernel includes no scheduling and synchronization policy, nor recursive capability revocation. This means that user-level logic must define policy on top of the kernel's policy-less system calls to dispatch to a thread and to modify capability-tables for already accessible resources.

As a result of these complex "raw" kernel APIs, both Patinas build extensive abstractions to hide as much of this complexity as possible. Their fundamental designs focus

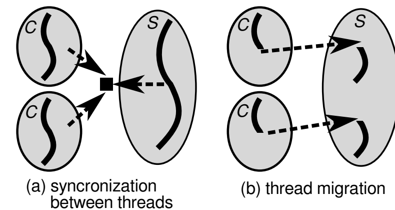


Fig. 2: IPC via synchronous rendezvous between threads (*e.g.,* L4) and via thread migration.

on principles and techniques to build from primitive system abstractions and APIs, up to a full-featured RTOS.

Composite Patina Design. Patina on Composite relies on two existing libraries, one for transparently creating and constructing the capability- and page- tables for protection domains (*components* in Composite), and another to easily use the dispatching and timing facilities of the kernel to create and schedule threads. On top of this, we add a library (the Composite runtime, or cosrt) to allocate and manage the capability and page tables of both the current component, and other components it creates, complete with the ability to load a component from an ELF file. The system abstractions at this point are still low-level and based on kernel resources – threads, IPC endpoints, pages, and components.

Composite focuses on predictable execution through component composition. Components are functionally composed together using the main Composite mechanism for IPC: synchronous thread-migration-based *invocation* [29]. Figure 2(a) depicts L4-style IPC using blocking rendezvous on an IPC endpoint between threads. Figure 2(b) shows thread-migration, in which, when a thread in a client component (C) invokes a function in a server component (S) in a separate protection domain, it executes using the *same schedulable entity* in both C and S. This maintains strong memory isolation as the execution context (registers and stack [30]) are separate for S and C. For invocation arguments and return values, a set of registers are passed and returned. Thread migration is integral to the Composite Patina implementation as it ensures a thread's execution maintains the same end-to-end real-time

TABLE I: Patina API

API area	API Functions	Description
Process and Thread Management	`process_create(), process_exit(), process_get_exit_status()` `thread_create(), thread_set_params(), thread_kill(),` `thread_exit(), thread_get_exit_status()`	Create processes and threads, terminate them, configure them, and retrieve exit status codes
Channels	`channel_create(), channel_destroy(), channel_get_recv(),` `channel_get_send(), channel_retrieve_recv(),` `channel_retrieve_send(), channel_close(), channel_send(),` `channel_recv()`	Create channels that can be either "named" or "unnamed". These channels have dedicated send and receive sides that must be explicitly opened or retrieved. These sides, then, allow sending or receiving
Timers and Time	`timer_precision(), timer_create(), timer_free(),` `timer_start_oneshot(), timer_start_periodic(),` `timer_cancel()` `time_current(), time_create(), time_add(), time_sub()`	Oneshot and periodic timers that can be canceled. API also exposes the current time and provides functions to manipulate time values
Synchronization	`semaphore_create(), semaphore_destroy(), semaphore_take(),` `semaphore_try_take(), semaphore_give()` `mutex_create(), mutex_destroy(), mutex_lock(),` `mutex_try_lock(), mutex_unlock()`	Standard semaphores and mutexes with take/lock, try_take/try_lock, and give/unlock operations. Mutexes support priority inheritance and (optionally) recursive locking
Event Handling	`event_create(), event_delete(), event_add(),` `event_remove(), event_wait(), event_poll()`	Create/delete event handlers, add or remove event sources, and wait or poll for events. Event sources include timers (fired), channels (ready to receive, ready to send), processes (exited), and others
Memory Management	`mem_alloc_pages(), mem_free_pages(),` `mem_shared_create_named(), mem_shared_destroy_named(),` `mem_shared_map_named(), mem_shared_create_anon(),` `mem_shared_destroy_anon(), mem_shared_map_anon()`	Allocate and release pages of memory as well as create both "named" and "anonymous" shared memory regions and map them into processes
I/O	`io_print()`	Output to a shared UART or console

execution properties as it would if it were executing in only a single component. This has a very important side effect: a *scheduler* component must define the blocking and synchronization policies. The scheduler's data-structures, logic, and policy define CPU allocation and synchronization.

For synchronous IPC, including thread-migration-based invocations, C's execution is tied to S's as C won't reactivate until S returns. Thread migration ensures that schedulers maintain a consistent scheduling context (priority, budget, *etc.*) while executing across the system. However, this poses a challenge: shared-resource access within S must be synchronized between client requests (as in Figure 2(b)). As a result, *the* blocking API that Composite schedulers export is designed to integrate predictable resource-sharing protocols by default. The combination of thread-migration-based IPC and efficient, predictable synchronization enables *local reasoning* about *full-system predictability*. Patina components implement their functionality following traditional real-time system principles: ensuring bounded execution, and sharing resources, without explicit consideration of the *composition of components*.

seL4 Patina Design. We implement a Patina on top of the seL4 kernel to take advantage of the integrity and confidentiality guarantees seL4 provides. In particular, seL4's verification ensures that data can only be read or written with permission and that the kernel implements its specification correctly [4], [31]–[34]. These powerful guarantees eliminate entire classes of bugs including memory-safety issues, undefined behavior, missing permissions checks, and even logic bugs.

Since seL4 provides only a complex API consisting of isolation, scheduling, and communication primitives, our Patina implements a set of user-space services and abstractions to simplify key operations. Each of these services consists of a thread in its own protection domain, including both capabilities and virtual memory. Note that protection domains are fundamentally *processes* in our seL4 Patina. Two services are central to the entire rest of the system: the loader service and the capability service. The loader service handles the creation and management of processes and threads, including transparently constructing and configuring capability tables, page tables, and thread objects and providing the ability to load a process from an ELF file. The capability service manages unallocated memory for other parts of the system, holding all untyped memory in the system, and allocating capabilities from this memory for the rest of the system. The rest of the Patina is implemented as a number of services, or dedicated processes, that build on these two core services, each providing a different aspect of the API, like events, timers, or channels.

Unlike Composite, each thread in seL4 is bound to its protection domain and communication is performed via IPC, where the sending thread is blocked and the receiving thread made runnable. In particular, seL4 IPC is rendezvous-style IPC and thus synchronous and blocking. Additionally, the scheduling of threads and control over blocking is baked into the seL4 kernel and not configurable by user space (as shown in Figure 1c). While this makes reasoning about full-system predictability more complex, the seL4 kernel is designed with a fixed-priority scheduler to enable real-time performance.

B. *Patina API Overview*

In this section we present an overview of our Patina API, summarized in Table I, emphasizing its expressiveness, while also touching on its implementation in Composite and seL4.

Timers. Timers enable time-triggered activations and can be one-shot or periodic. Timer activation occurs in the form of an event that will be delivered through the event-handling API.

In Composite, user-level schedulers have the ability to program one-shot timers (within their TCap budget [35]), thus allowing the scheduler to implement timers and control preemption. The timer manager tracks Patina software timers, and triggers expired timer events via the event component. In seL4, we implement a timer service that uses a dedicated hardware timer to generate interrupts. This timer service manages a timer wheel to track software timers and communicates with the event service to generate timer events.

Channels. Channels provide buffered data transfer of messages between endpoints that may be in separate processes. Channels may be either named, allowing them to be addressed globally, or unnamed, requiring them to be shared explicitly. By default, read and write operations are non-blocking, but blocking reads and writes may be optionally implemented. This default behavior avoids inter-application synchronization and encourages blocking awaiting multiple notifications.

In Composite, channels are implemented using a shared-memory wait-free message queue to avoid blocking synchronization. The channel manager sets up and tears down these channels while a library provides the message-queue implementation. In seL4, channels exist in a dedicated channel service and all read/write operations are performed as IPC messages to this channel service.

Event Handling. The Patina event-handling API enables a caller to be edge-notified of one or more events in either a blocking or non-blocking manner. Events are generated by other Patina resources in response to events (*e.g.,* a timer firing). By adding one or more of these resources to an event handler, a thread can wait for events on those resources, much like the `select()` and `epoll()` system calls.

In Composite, a dedicated event-manager component hands event-notification endpoints to event listeners and event-triggering endpoints to event sources. The event manager ensures event ordering. In seL4, a dedicated event service hands notification endpoints to event listeners. Event sources perform an IPC to this service to trigger an event.

Synchronization. Patina provides synchronization in the form of both mutexes and semaphores. For predictability, Patina mutexes support priority inheritance (PI) [36].

As Patina currently focuses on single-core systems,[3] both Patinas provide blocking-synchronization variants, rather than spin-based. Composite exposes a scheduler-provided abstraction for blocking that decouples fast-path (uncontended lock) access, from blocking, similar to Futexes [37], [38] (see synchronization in both an application library and service in Figure 1(b)). The seL4 Patina uses a separate synchronization server that leverages the client's blocking IPC to halt the thread requesting a lock, while replying only to the highest-priority blocked thread to allocate the lock. We discuss synchronization in the seL4 Patina in greater detail in §IV-D.

[3]Mainline seL4 does not include verified multicore support.

Thread Management. The Patina execution abstraction is threads, and conventional (`pthread`-like) APIs for setting parameters, exiting, and joining on them are supported. In our Composite Patina, this is implemented in the scheduler while our seL4 Patina implements it in the loader service.

Memory Management. Memory can be dynamically allocated and released and shared memory is supported. Shared memory may be either named, allowing it to be addressed globally, or unnamed, requiring it to be explicitly shared.

In Composite, static memory (data and bss, read-only data, code, *etc.*) is provided at boot time by the constructor component, which is responsible for creating not only application components, but also the service components, and does not expose APIs for application interaction. After boot, the capability manager is in charge of providing dynamic allocations, and exposes memory-management APIs, including those for shared memory. In seL4, the kernel sets up memory for the initial loader and capability services. All dynamic memory after that point is allocated by the loader service, in collaboration with the capability service. In particular, the loader service exposes memory-management APIs, including those for shared memory, to applications.

C. PoLP Design in the Composite Patina

Here we explain the primary mechanisms by which we support and enforce least privilege in the Composite Patina, while providing efficient and predictable functionality.

Authority decentralization in the Composite Patina. Authority is distributed throughout the components of the system as shown in Figure 1(b) by applying the separation of concerns to break the system software into pluggable, mutually isolated components, each responsible for different resources.

The Composite Patina adds a service component for each abstract resource: a *channel manager*, *event manager*, *timer manager*, and *scheduler*. Service components that manage kernel resources have access only to the subset of appropriate resources. These include the *scheduler* (that dispatches threads), the *capability manager* (that defines delegation and revocation policies), and the *constructor* (that creates/loads the graph of components). This has the benefit that key components relied on by many others focus on simplicity. The PoLP guides the design by enabling only the scheduler to dispatch threads, only the constructor to have access to the static memory allocations of each component (code and data), and only the capability manager to have access to untyped memory for dynamic allocation to other components. Figure 1(b) shows how capability-management policy is distributed between (1) process creation in the constructor, and (2) dynamic management in the capability manager.

Components cannot alter their capability access and instead rely on the capability manager to pass resources and revoke access to them. In contrast to L4-style μ-kernels that define capability delegation and revocation policies in the kernel, the capability manager defines the dynamic capability delegation and revocation policies for kernel resources.

The constructor is the only component created by the kernel at boot-up, and it is responsible for loading all other components. It starts with access to all system kernel resources (*i.e.,* all memory) and distributes them among components based on a static specification of components and their dependencies. Importantly, the constructor creates the initial component images (including all non-dynamic memory) and the initial set of capabilities. Thus, only the constructor has access to static component memory, decoupling this static privilege from the dynamic memory and resource management in the capability manager. The constructor also creates the synchronous invocation capabilities that enable invocations between components. A side effect of this is that the inter-component control flow (*i.e.,* the control flow between components) is constrained solely by the constructor, providing a form of inter-component Control Flow Integrity [39] (CFI). To strengthen this CFI, after initialization of the capability manager, the constructor is not executed again (aside from for faults).

System simplification via custom resource management. As Composite components can be tailored to a specific set of requirements, we focus on economy of mechanism to implement Patina. Though §**??** discusses this quantitatively for all services, below we discuss three examples.

First, *blockpoints* are the *only blocking abstraction* in Composite and are provided by the *scheduler*. A blockpoint is similar to a condition variable in that it enables threads to *block* or to *wake up* a single thread or all threads blocked on a blockpoint. However, *unlike* condition variables, they do not require mutexes, and are instead intended to work with lock-free data structures. Indeed, the implementations of *mutexes*, *semaphores*, and *channels* require blocking synchronization. Each of the data structures that back these abstractions use atomic instructions to coordinate (*e.g.,* to set the *owner* of a mutex with a *compare-and-swap* instruction) and integrate with blockpoints as follows:

1) repetitively execute the following,
2) take a *checkpoint* of the abstraction's blockpoint,
3) update the data structure atomically, and if we do not need to block (*e.g.,* we take the critical section or can dequeue from a channel), break out of step 1's loop,[4]
4) otherwise *block* on the abstraction's blockpoint.

Another thread can wakeup others blocking on the blockpoint by later *triggering* the blockpoint. The "lost wakeup" race condition motivated the creation of blockpoints. If preemptions lead to the trigger happening between steps 3 and 4, we have a lost wakeup, and the blocking thread might never awake.

Blockpoints avoid this race condition by separately tracking a blockpoint *epoch* in the library, and in the scheduler. Operations performed on the blockpoint increment the epoch, thus the scheduler can detect lost wakeups as the epoch passed with the operation will be less than that in the scheduler.

[4]Note that, despite the "retry loop," a thread will execute the retry loop at most once per higher-priority thread that changes the state of the backing resource. Thus, to ensure predictability, the small overhead of a retry can be accounted for similar to context switch costs in a timing analysis.

Blockpoints also express *dependencies* between threads. When one thread blocks, it can express that it is waiting for (dependent on) another (*e.g.,* a mutex holder). This enables the scheduler to perform PI properly.

The blockpoint API aims to solve a similar problem to that solved by Linux Futexes [37], [38]: providing fast, library-based coordination when blocking is not necessary and a means to avoid lost wakeups when blocking is necessary. Blockpoints do so with significantly less complexity by identifying each blockpoint with an opaque id rather than a *physical address* and not requiring that the scheduler access the blockpoint memory. The result of this intentional design is that the scheduler's blockpoint implementation is only 103 C Lines of Code (LoC), with the client library being another 105 LoC, while `futex.{h,c}` are over 1850 LoC and intertwined with the virtual memory subsystem. Customizing blockpoints to the requirements of Patina avoids the PoLP-violating intertwining of virtual memory and scheduling while maintaining strong average-case performance.

Second, the capability manager defines resource-access delegation and revocation enabling it to be vastly simplified by designing explicitly for the limited sharing relationship of Patina. Traditional (in-kernel delegation/derivation) structures track all delegations (and retypes) in a tree, and recursively remove a subtree of delegations on revocation. Channels use shared memory between two applications, which requires page allocation and two delegations. The Composite Patina specializes the data-structure that tracks resource delegations by statically allocating it based on the maximum number of allowed delegations. The simplicity of this implementation – the capability manager's logic is less than 700 LoC – avoids dynamic memory allocation, has only bounded loops, and enables the use of a lock-free structure to avoid mutex-based synchronization. This is important as the very lowest-level components cannot leverage the services of the scheduler.

Third, channels in the Composite Patina use memory shared directly between applications. We arrived at this design after assessing three different channel implementations. A design constraint is that Composite IPC passes only a register-set between components with a synchronous invocation. The first design passes all channel data to the channel manager using many invocations, each passing a few words of data. This design is simple and does not require shared memory, but is slow due to the many invocations. The second design uses shared memory between client channel libraries and the channel manager to pass data. This design trades simplicity for performance and centers trust in the channel manager.

Our final design uses direct shared memory for passing data between applications. This has the benefit of removing the channel manager from fast-path operations. Toward the PoLP, this design vastly simplifies the manager as it provides only channel setup and tear-down. However, it does expose applications to mutually shared memory, which is a *wide* interface that requires a complex functional correctness analysis.

The shared memory is used only for a bounded, static, wait-free ring buffer and uses no pointers. All library accesses

to the ring buffer are explicitly bounds-checked to prevent errant accesses, and data is copied in and out of the buffer so, outside of the channel code, data access is only to non-shared memory. However, malicious applications can directly modify any buffer entry along with the `head` and `tail` offsets. Note that maliciously modifying data in the buffer is in many cases equivalent to normal API operations (*e.g.,* sending corrupted data). There is one exception: a compromised application can corrupt messages in the channel that were sent before compromise but have not yet been received. The Composite Patina accepts this risk as receivers must sanitize and validate channel messages they receive regardless, and thus must handle a broader range of corrupted messages.

As recoverability of the system in the face of adversaries is a core design goal of Patina, all manager components track allocations made to other components. A fault or compromise in an application requires that each service be notified of the failure, at which point it can reclaim all associated resources.

Predictability in the Composite Patina. The Composite Patina is focused both on predictability and on minimizing task interference. Thread-migration-based IPC is the enabling feature. Figure 2(b) demonstrates that invocations to a service component S are conducted in the same scheduling context as executed in C. Thus, all execution in S is prioritized to that of the client thread, but must consider concurrent client requests.

The Composite Patina uses two techniques to avoid this interference: (1) most service components have simple data structures in which new abstract resources (*e.g.,* channels and events) are created, but then not modified (or modified using only wait-free structures) using the techniques in parsec [40], [41], thus avoiding the need for mutual exclusion, and (2) for the timer and scheduler services that require more complex structures (runqueues, and timeout heaps), we use predictable mutexes with PI provided by efficient, blockpoint-based locks. All paths in all services are carefully engineered to be bounded (no unbounded loops, no recursion) to support WCET reasoning. We also avoid all nested locking in the system components so a PI-aware timing analysis is straightforward.

D. PoLP Design in the seL4 Patina

The seL4 Patina also seeks to apply the PoLP to its design, although how that occurs in practice differs significantly from the Composite Patina. In this section we examine the primary design decisions for PoLP in our seL4 Patina.

Authority decentralization in the seL4 Patina. Authority in the seL4 Patina is distributed throughout the components of the system as shown in Figure 1c by applying the separation of concerns to break the system into separate protection domains (*i.e.,* processes) each responsible for different resources. Much like the Composite Patina, there is one service process per Patina resource: a *channel service*, *event service*, *timer service*, and a *synchronization service*, as well as a *loader service*, which is responsible for processes, threads, and memory.

Our seL4 Patina also decentralizes authority by introducing *self-contained processes*, a novel feature found in no other system we are aware of. To be self-contained means that no

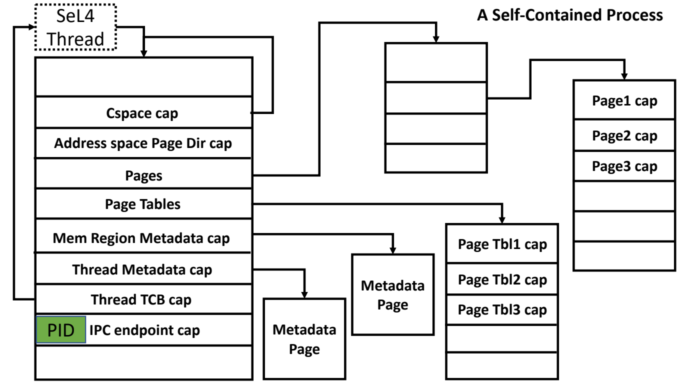


Fig. 3: Self-Contained Processes in the seL4 Patina

other part of the system has access to a process's capabilities, memory, or identity, including the memory-management code, process-management code, and loader. Other parts of the system can only access this information if the process explicitly requests some operation be performed. This design minimizes privilege by ensuring that no single element of the system has access to all memory or all capability tables or all threads in the system (outside of the formally verified seL4 kernel).

Self-contained processes are valuable for high-reliability systems because they reduce the trusted computing base (TCB) once a process is running and help ensure the ability to recover from failures. In particular, they ensure that even the component that created a process, initialized its address space, or provided its capabilities cannot subsequently modify those capabilities. Thus, once a self-contained process is running and has allocated sufficient memory, it can continue running without needing to place future trust in other elements of the system. This is in contrast to monolithic kernels (*e.g.,* Linux) in which the kernel still retains access to all memory and can arbitrarily remap or unmap memory out from under a process.

The development of self-contained processes was possible primarily because seL4 does not have a process abstraction, but only threads, address spaces, and capability spaces. As a result, we were free to design a process abstraction from scratch. This abstraction consists of one or more threads, a virtual address space, a capability space, metadata, and identity, as depicted in Figure 3. The core of a process is its capability space, which both contains a capability to itself and contains capabilities for all the other elements of the process. It is by ensuring that the only capability to this capability space is in itself that the seL4 Patina ensures self-contained processes. Many capabilities are required for virtual memory, including page-frame capabilities, page-table capabilities, and page-directory capabilities, as well as capability-table space to store all of these capabilities. This is complicated by the fact that seL4 capabilities allow the owner to modify an object but not query it,[5] necessitating metadata, stored on more pages, to determine where page capabilities are mapped, *etc.* Identity is achieved by badging all IPC endpoints in the process's capability space – the only way it can communicate with the

[5]This comes from the information flow guarantees of the seL4 kernel; the ability to query configuration would expose a trivial storage side channel [32].

outside world – such that the process's process ID will be unforgeably conveyed with any IPC messages.

Restoring minimal control via escrow processes. While self-contained processes have many benefits, they have one major downside: they allow malicious processes the same strong isolation guarantees afforded other processes. In particular, a self-contained process can resist attempts to terminate it, can prevent its resources from being reclaimed, and can modify its capability space or metadata and then request operations requiring the seL4 Patina to parse these structures. Since seL4 capabilities cannot be re-identified once moved, the seL4 Patina has a way to assert limited control over untrusted processes that may be, or could become, malicious.

This limited control consists of two parts: (1) the ability to force controlled termination of an untrusted process, and (2) the ability to prevent untrusted processes from modifying their capability space or metadata arbitrarily. Note that this is much narrower than the modifications that can be made to any process in traditional systems; in particular, memory maps cannot be changed nor can capabilities be added or removed.

The seL4 Patina provides this control using *escrow processes*. An escrow process is a trusted process whose sole role is to manage the capability space (and all the capabilities and metadata stored inside) of an untrusted process. In essence the escrow process hold a process's capability space "in escrow" making any required modifications on behalf of the process while ensuring that those modifications are done correctly and providing a way to command the process. Note that there is one escrow process for each untrusted process to avoid centralizing these capabilities in a single service.

The escrow-process executable is part of the core of the seL4 Patina and is a self-contained process itself. It is designed to hold the capability space of a single untrusted process and perform all legitimate modifications of that capability space on behalf of that process. As a result, requests to other services that require a process's capability space must first be sent to the escrow process, to get the capability space, and then sent on to the desired service. Examples of operations requiring the capability space include: starting a new thread, allocating memory pages, and creating an event handler.

Interface complexity. Because the threat model for Patina includes malicious processes, as discussed in §II, the internal interfaces between components are particularly important, as they represent attack surface. A variety of different designs for these interfaces are possible, with varying levels of complexity and efficiency. However, a particularly important concern is how well the interface maintains correct behavior in the face of a malicious party. This is often tied to the complexity, or wideness, of the interface: wide interfaces tend to have shared state and implicit protocols about how to update that state while narrow interfaces tend to have partitioned state and explicit protocols about how to communicate state changes.

As an example, consider a correct process communicating over a buffered message queue (*e.g.,* a Patina channel) with a compromised, malicious process. As discussed in § IV-C, a shared memory implementation contains a wide interface with a number of variables representing the state of the channel. This state needs to be updated using an implicit protocol that the adversary is free to ignore by, for example, placing one message in the queue but claiming to have placed 100 messages in the queue. The code to check for and safely handle these kinds of issues is complex and frequently incorrect, leading to vulnerabilities. Our Composite Patina frequently uses these wide, shared memory interfaces and then provides extensive correctness analysis to ensure safety.

In contrast, our seL4 Patina relies on narrower interfaces that can take advantage of the seL4 kernel's formally verified IPC path to ensure safety from malicious parties. In particular, the seL4 kernel supports IPC messages up to about 480 bytes, by using a special memory page as an IPC buffer, and this implementation is formally verified. For example, in the seL4 Patina, channels are implemented by storing the message queue in the channel service (inaccessible to processes directly) and using seL4 IPC to communicate with the channel service to send and receive messages. This places the state in the channel service, keeps the data communicated via IPC incredibly simple – operation type, channel id, and message data – and maximally leverages the kernel's formal verification. On the other hand, it does introduce more overhead – several additional data copies and two instead of one IPC calls – compared to a shared memory approach. This kind of architecture is also used for mutexes, event handling, *etc.*

Predictability in the seL4 Patina. The seL4 Patina also places emphasis on predictability. However, its ability to provide predictability and minimize interference among tasks is constrained by the seL4 kernel, which is opinionated about scheduling and blocking, defining a rigid policy in the kernel.

seL4 only provides a fixed-priority scheduler and provides no mechanism to determine a thread's current priority. Thus, we statically prioritize services over user applications. The service priorities are carefully chosen, with services that interact with hardware the highest, followed by the event service, loader, capability service, and the other Patina services.

Recent versions of seL4 [42], [43] support execution-time budgets and the ability to donate part of a thread's budget to another thread. We did not use these extensions in our seL4 Patina, because without thread migration attaching a budget to a service risks that service exceeding its budget early, starving other higher-priority requests. Donating time from requestors does not solve this problem, because there is not certainty that a requestor's time will go towards its own request.

We also explored the creation of a user-level scheduler, as demonstrated in [44], that would manipulate thread priorities to effectively control scheduling. Unfortunately, all designs we were able to devise resulted in the centralization of all thread capabilities and required the user-level scheduler to interpose on nearly all system calls and service interactions, doubling the amount of IPC and corresponding overhead.

Adding PI support for mutexes in the seL4 Patina synchronization service was complicated. seL4 has notification capabilities that appear to be well suited for synchronization, with a wait operation that blocks a thread and a signal

operation to unblock a single waiting thread. Unfortunately, the wait-queue design is FIFO, not priority based, and does not support PI. As a result, we developed an alternative blocking mechanism for mutexes that enables PI. This mechanism leverages the IPC reply capability generated by a two-way IPC call. Essentially, mutex lock and unlock operations become IPC calls to the synchronization service, which does not reply to the IPC, releasing the thread, until that thread owns the mutex. To provide PI, a copy of each thread's thread capability must be supplied to the synchronization service prior to the first lock operation by that thread. Then, when a higher-priority thread blocks on a mutex, the service can increase the owning thread's priority temporarily using its thread capability.

V. Evaluation

In this section, we evaluate both Patina implementations to characterize their performance and predictability. In particular, in our evaluation, we seek to: (1) assess the latency of time-triggered activations using the Patina API for real-time computation, (2) evaluate the performance and predictability of Patina operations with functionality that spans multiple, isolated services, and (3) use Linux with the `PREEMPT_RT` patch as a baseline for a system with strong average-case performance, and, in many domains, acceptable predictability. These results should enable us to ascertain if systems designed for the PoLP can achieve strong, predictable performance.

A. Methodology and Experimental Setup

For our evaluation, we use the popular Zynq-7000 XC7Z020 SoC, which includes a dual-core Arm Cortex-A9 processor running at 667 MHz and a Xilinx FPGA. We use only a single core for this evaluation, and do not use the FPGA at all. We use gcc version 8.3.0 (Debian 8.3.0-2) for arm-linux-gnueabi-gcc and evaluate against Linux kernel version 5.4.61-rt37. Our seL4 Patina was built with `rustc` version nightly-2020-05-31. All systems use the built-in UART to output results.

Unless otherwise noted, each result is computed from 10,000 test runs. In our seL4 Patina, the user-level timer device backing the timer manager is disabled to avoid interference (on runs that do not use the timer), though the kernel's timer is not modified. In our Composite Patina and in Linux, we avoid using timers, but do not disable the kernel timer, thus timer interference is present in some results. We take many samples so that the impact of this interference is minimized, though the maximum measured readings likely include its impact. We filter out the first sample on all systems.

B. Analysis

Table II summarizes our results, and Fig. 4 shows Cumulative Distribution Functions (CDFs) for Patina operations: mutex locking, timer expiration, and channel communication.

Core System Overheads. Each system exhibits core overheads for system operations such as thread context switches. Additionally, IPC overhead in both μ-kernels is critical, as Patina functionality is provided by services that are composed using IPC. These overheads are important to understand the overheads of different Patina functionalities.

Discussion. Both μ-kernels have IPC on the order of Linux system calls (measured with `close(999)`), which demonstrates a basic feasibility of a multi-process, PoLP-focused system. Native seL4 round-trip IPC takes 660 cycles, so the seL4 Patina which includes serialization and deserialization adds only around 50% overhead. seL4's thread switch latency is quite low (almost an order of magnitude faster than Linux's), and has tight bounds. Composite IPC is faster than seL4's, but thread switches through the user-level scheduler incur more overhead. Note that Slite [6] removes many of these overheads by avoiding kernel interactions on thread switches, but we have not ported it to this platform yet. Further points of comparison are available through data gathered from other common RTOSes, such as QNX [45].

We also provide extensive comparisons between Linux and the two implementations of Patina in Table II. In particular, we compare Linux against not only equivalent Patina operations, but also, in the top row of Table II, against the raw μ-kernel performance for context switching and IPC. These system outputs quantitatively demonstrate performance with and without Patina support. As these metrics are the basic building blocks for more complex system components, they are fitting as core comparison values.

Event handling is a core operation in the Patina API. To evaluate its performance, we add a debugging API to allow applications to trigger events. We measure the latency between this trigger and when the event-wait operation returns. This gives us an indication of how much overhead the event subsystem adds to the other measurements. There is no Linux equivalent of this measurement, as there is no direct way to raise an event, thus all means of measurement would also include another system abstraction (*e.g.,* writing to a pipe).

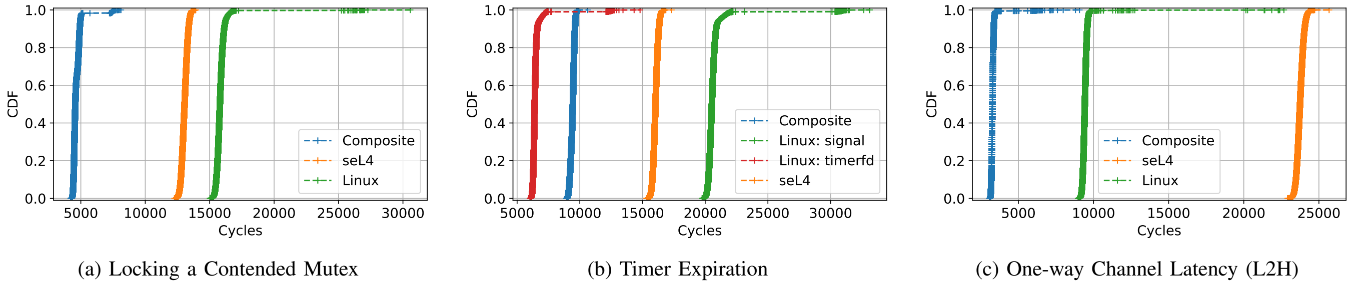
Channels. Patina provides sized channels for communication between processes. Here we evaluate the latency from when a message is sent to when it is received, both for the case when the sender is higher priority than the receiver, and when it is lower. Note that the seL4 Patina does not implement the optional blocking channel API. In Linux, we evaluated both pipes and sockets (both UNIX Domain sockets and UDP sockets) and concluded that pipes have the least overhead, so we compare Patina channels against Linux pipe overheads here. Higher-priority senders uniformly exhibit more overhead as they must block to execute the low-priority receiver.

Discussion. The average overhead of the Composite Patina is *less* than that of Linux, and the measured worst case costs for channel operations for the seL4 Patina are close to those in Linux. These results show that PoLP-based Patina implementations can be competitive with Linux.

Timers. Awaiting a timer expiration in Patina involves the timer device, the timer manager, and the event manager to convey the timeout event to the application. In Linux, we evaluate multiple methods for measuring timer-propagation latency, including using signals with a handler that simply writes into a pipe (the common, re-entrant means of handling signals) that is read by a target thread, and using a `timerfd`

TABLE II: Patina Overheads in Cycles in Composite and seL4 with equivalent Linux operations.
† the seL4 Patina does not implement the optional blocking channel API. * No direct Linux equivalent.

	Linux				Composite Patina				seL4 Patina			
	Avg	Std Dev	95%tile	Max	Avg	Std Dev	95%tile	Max	Avg	Std Dev	95%tile	Max
Context Switch: Thread	1,060	25	1,077	3,232	959	158	978	7,474	542	12	563	597
Context Switch: Process	4,816	327	4,858	17,919	1,617	174	1,630	7,888	542	12	564	703
Round Trip IPC	*	*	*	*	540	3	543	733	989	19	1,027	1,113
Event Latency: equal prio	*	*	*	*	2,868	203	2,954	9,114	11,504	175	11,801	12,247
Event Latency: L2H prio	*	*	*	*	2,883	217	2,970	9,124	11,407	176	11,702	12,233
Event Latency: H2L prio	*	*	*	*	2,843	212	2,930	9,002	16,585	222	16,953	18,160
Mutex Uncontended	217	2	217	328	125	61	126	4,196	9,959	184	10,270	11,165
Mutex Contended	15,844	619	16,263	30,570	4,677	412	4,974	8,116	13,053	234	13,440	13,918
Semaphore Uncontended	116	90	116	9,112	104	35	104	3,558	9,051	179	9,357	9,792
Semaphore Contended	6,713	404	6,994	22,136	4,597	382	4,880	8,186	11,430	217	11,791	12,384
Timer Latency	20,665	1,068	21,171	33,118	9,422	159	9,654	10,630	16,042	203	16,381	17,317
Timer Latency w/ timerfd	6,493	632	6,842	14,806								
Channel Latency: L2H prio	9,439	423	9,627	22,671	3,290	243	3,388	9,066	23,749	230	24,138	25,678
Channel Latency: H2L prio	11,507	841	11,711	71,169	4,086	234	4,194	10,536	24,839	229	25,222	27,806
Channel: L2H, direct blocking	6,440	346	6,594	19,321	2,351	185	2,426	6,480	†	†	†	†
Channel: H2L, direct blocking	9,408	1,013	9,591	92,286	2,572	196	2,648	7,622	†	†	†	†



(a) Locking a Contended Mutex

(b) Timer Expiration

(c) One-way Channel Latency (L2H)

Fig. 4: Cumulative Distribution Functions (CDFs) for three key Patina operations compared to equivalent operations on Linux.

to direct the timer event to a file descriptor. The former is a POSIX-compliant approach, while the later is Linux-specific. In both cases, we use `epoll` to await the event. To measure the entire timer-propagation latency, we use a low-priority thread that simply spins saving a cycle count into a global variable. A higher-priority thread is notified by the timer and immediately retrieves a cycle count and compares to the global variable. In both Patina variants, multiple processes are executed. In the Linux variant, only a single process is involved, thus the results favor Linux.

Discussion. Average Linux timer latency is low when using Linux-specific APIs. However, measured maximums indicate a significant variance of execution times. In this case, Linux does not show significant benefit over the seL4 Patina while the Composite Patina demonstrates overhead improvements.

Synchronization. Measuring uncontended mutex and semaphore latency is straightforward (when using a single thread and with semaphores initialized to a positive value). Contention is more involved to measure, but possible with a little careful effort: a low-priority lock holder activates a higher-priority contender, and the priority-inheritance-assisted boosting and eventual switch back to the higher thread is measured. We are careful in all cases to measure no additional APIs or behaviors other than lock contention.

Discussion. Both Linux and the Composite Patina successfully use mechanisms (Futexes and blockpoints, respectively) to avoid system calls in uncontended cases. Due to the general complexity of the contended-case Linux code that uses PI, the measured maximum overheads (the main consideration in a schedulability analysis) eclipse that of either Patina. Mutexes that do not support PI on Linux demonstrate an overhead of around 7200 cycles, so there is a significant cost to predictability. Note that the maximum overheads in the Composite Patina for uncontended mutexes and semaphores demonstrate that we do not filter out timer-tick processing in the results – around 3000 cycles of overhead.

TABLE III: Lines of Code for the Patina implementations.

	Composite (C code)	seL4 (Rust code)
Event Handling	249	1016
Channels	688	1911
Timers	191	1681
Sched/Synchronization	2569	1863
Memory/Cap Management	696	8110
Core System/Libraries	5175	5075
Kernel	9227	(C code) 9300
Total	18795	29056

Complexity. The number of Lines of Code (LoC) in each Patina implementation is depicted in Table III, though this is an imperfect complexity metric. The higher-level RTOS functionality increases the LoCs over native kernels, but remains much simpler than monolithic systems. Even the QNX Neutrino kernel v6.3.2, which provides similar functionality without PoLP-based isolation, is 23K LoC.

Summary. The results show that the measured overheads of our Patina implementations do not suffer overheads significantly greater than Linux. In many cases, the Patina implementations demonstrate performance better than Linux. We believe this

demonstrates that a PoLP-based Patina design is a reasonable and appealing direction for high-criticality embedded systems.

VI. Discussion

In this section, we reflect on our two Patinas, both built with a PoLP emphasis, but with different foci and restrictions. We discuss how these differences expose trade-offs in design and performance between the two implementations.

Policy defined by kernel vs user space. One of the major differences between our Patina implementations is that the Composite kernel pushes all policy, including scheduling and resource delegation and revocation, to user space. In contrast, seL4 defines scheduling and resource policies in the kernel.

seL4's choice to define policy in the kernel initially simplifies the system; no user-space scheduler is required before multiple applications can be run, for instance. However, resolving situations where seL4's policies do not provide what is expected for the Patina API can be complex. For instance, seL4 provides a notification capability that seems well-suited for creating mutexes and semaphores, but because it does not provide priority inheritance, our seL4 Patina had to take a different approach that was less efficient. This is a major reason that the seL4 Patina mutexes and semaphores do not have a fast uncontended case and why the Composite Patina mutexes and semaphores are faster. This mismatch between seL4 policy and Patina expectations also arises with respect to memory management and seL4's policy that capabilities cannot be used to query the current state of a capability. This results in our seL4 Patina needing to track memory and capability metadata separately, which requires over $8,000$ lines of code, compared to the under 700 lines required by the Composite Patina, as shown in Table III.

Placing scheduling policy at user-level enables timing-policy customization and constrains the access of the scheduler to that appropriate for scheduling (consistent with the PoLP). However, this imposes overheads for scheduler-component invocations. The Composite Patina demonstrates increased context-switching overheads over seL4, but, interestingly, similar magnitude overheads to Linux. This demonstrates the practicality of user-level scheduling.

Analysis-simplicity vs. performance-focused designs. In §IV-C, we discussed an analysis of the functional correctness and the impacts of a compromise on the channel implementation in the Composite Patina. This is trade-off made by the two Patina implementations. The seL4 Patina uses the kernel's facilities for passing data along with IPCs and uses the channel manager to control all channel logic. In leveraging the kernel's verified paths for copying a fixed, bounded data amount, this implementation focuses on high confidence. The downside of this approach is in overhead, as shown in §V-B.

In contrast, the Composite Patina uses shared memory for data movement between communicating applications. This improves performance compared to Linux. However, the shared-memory approach to data sharing complicates the functional-correctness analysis (the wide-API must consider any combination of loads and stores as discussed in §IV-C).

Predictability of Patina implementations. Despite their differences, both of our Patina implementations provide performance on par with Linux, if not better. This is unintuitive given the larger structural costs in our PoLP-focused Patinas due to isolation, and given Linux's strong emphasis on average-case performance. These results indicate that despite the focus on strong isolation and the PoLP, our Patina implementations demonstrate surprisingly competitive performance.

More importantly, the predictability of the Patina results is key for embedded and real-time systems. Both Patinas demonstrated very stable, predictable performance for key Patina functionality, with minimal tail latencies, as illustrated in Figure 4. Previous results have demonstrated that real-time predictability with competitive bounds can be achieved with user-level interrupt handling [5], even with a user-level interrupt-scheduling policy [17], and that user-level scheduling can have practically competitive performance [6]. We believe that we have advanced the arguments for security-focused RTOSes by demonstrating that the increased security and isolation from a multi-protection domain RTOS does *not* come at the cost of prohibitive overheads or higher latencies.

Benefit of Patina. The primary benefits of Patina is two fold. First, Patina abstracts the low-level API provided by μ-kernels. For instance, to create and start a new thread under seL4, capabilities must be created from untyped memory for memory such as the stack, and IPC buffer(s). Page directories and page tables must be created and managed, the scheduling priority must be set, and initial register values initialized. Composite exposes a similarly low-level API that also makes starting a thread a complex, multi-step operation. In contrast, Patina provides one call to handle this setup.

Second, this work argues that Patina implementations should be designed to separate the API implementation into many separate protection domains. While this introduces minor overheads, as illustrated in our evaluations, it decouples the different aspects of the API and prevents a fault in a single part of the API implementation from compromising all API calls across all applications. For example, a failure in the channel or event management services will not necessarily impact a high-criticality device driver. Isolation is also fundamental to being able to recover from such failures.

VII. Conclusion

We have presented the concept of OS Patinas, which provide feature-full OS abstractions on top of a μ-kernel. To demonstrate the feasibility and performance of OS Patinas, we independently implemented two Patinas, one on Composite and one on seL4, each guided by the PoLP. Past work has shown that shifting system services and scheduling policy from the kernel to user level can be implemented efficiently but this is the first attempt to apply the PoLP on the scale of an entire RTOS API. In exploring Patina designs on two separate μ-kernels, we have found that performance is comparable and in many cases even supersedes that of monolithic kernels. Our PoLP-based implementations also provide strong isolation.

REFERENCES

[1] M. M. Madden, "Challenges using linux as a real-time operating system," in *AIAA Scitech 2019 Forum*, 2019.

[2] "Mitre common vulnerabilities and exposures: http://cve.mitre.org/."

[3] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *in Proceedings of the IEEE*, vol. 9, no. 63, 1975.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[5] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *In Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[6] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: OS support for near zero-cost, configurable scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 160–173.

[7] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[9] M. Simos, "Overview of petya, a rapid cyberattack," Feb 2018. [Online]. Available: https://www.microsoft.com/security/blog/2018/02/05/overview-of-petya-a-rapid-cyberattack/

[10] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *Proceedings of the 2013 34th IEEE Real-Time Systems Symposium (RTSS)*, 2013, pp. 21–32.

[11] J. Shapiro, J. Smith, and D. Farber, "EROS: A fast capability system," in *17th ACM Symposium on Operating systems principles*. ACM, December 1999, pp. 170–185.

[12] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 26, no. 1, pp. 29–35, 1983.

[13] J. Liedtke, "On μ-kernel construction," in *15th ACM Symposium on Operating Systems Principles*. ACM, December 1995, pp. 237–250.

[14] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[15] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: Improving reliability through operating system structure," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[16] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software*, 2012.

[17] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[18] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[19] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov 2011.

[20] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther, "The sawmill multiserver approach," 2000.

[21] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The flux OSKit: A substrate for kernel and language research," in *Symposium on Operating Systems Principles*, 1997, pp. 38–51.

[22] G. Parmer and R. West, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *ACM Transactions on Software Engineering (TSE)*, July/August 2012.

[23] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, 2007.

[24] B. D. Fleisch and M. A. A. Co, "Workplace microkernel and os: A case study," *Softw. Pract. Exper.*, 1998.

[25] B. Leslie, "Grailos: A micro-kernel based, multi-server, multi-personality operating system," in *Workshop on Object Systems and Software Architectures (WOSSA 2006)*, 2006.

[26] D. R. Engler, F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[27] "The Fiasco microkernel and L4Re: http://l4re.org, retrieved 10/6/17."

[28] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010, pp. 209–222.

[29] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, 2010, p. 91.

[30] Q. Wang, J. Song, and G. Parmer, "Stack management for hard real-time computation in a component-based OS," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[31] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *International Conference on Interactive Theorem Proving*. Springer, 2011, pp. 325–340.

[32] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 415–429.

[33] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 471–482.

[34] T. Sewell, F. Kam, and G. Heiser, "Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–11.

[35] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 56–67.

[36] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[37] A. Zuepke, "Turning futexes inside-out: Efficient and deterministic user space synchronization primitives for real-time systems with IPCP," in *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[38] A. Zuepke and R. Kaiser, "Deterministic futexes: Addressing WCET and bounded interference concerns," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 65–76.

[39] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[40] Q. Wang, T. Stamler, and G. Parmer, "Parallel sections: Scaling system-level data-structures," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–15.

[41] Y. Ren and G. Parmer, "Scalable data-structures with hierarchical, distributed delegation," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 68–81.

[42] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–16.

[43] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[44] M. Åsberg and T. Nolte, "Towards a user-mode approach to partitioned scheduling in the seL4 microkernel," *ACM SIGBED Review*, vol. 10, no. 3, pp. 15–22, 2013.

[45] Hildebrand, Dan, "An Architectural Overview of QNX," in *USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992, pp. 113–126.